Polymorphic Functions with Set-Theoretic Types - Part 1: Syntax, Semantics, and Evaluation

Polymorphic Functions with Set-Theoretic Types

Part 1: Syntax, Semantics, and Evaluation

Giuseppe Castagna[1] Kim Nguyễn[2] Zhiwu Xu[1,3] Hyeonseung Im[2] Sergueï Lenglet[4] Luca Padovani[5]

[1]CNRS, PPS, Univ Paris Diderot, Sorbonne Paris Cité, Paris, France [2]LRI, Université Paris-Sud, Orsay, France
[3]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[4] LORIA, Université de Lorraine, Nancy, France [5]Dipartimento di Informatica, Università di Torino, Italy

Abstract. This article is the first part of a two articles series about a calculus with higher-order polymorphic functions, recursive types with arrow and product type constructors and set-theoretic type connectives (union, intersection, and negation). In this first part we define and study the explicitly-typed version of the calculus in which type instantiation is driven by explicit instantiation annotations. In particular, we define an explicitly-typed λ-calculus with intersection types and an efficient evaluation model for it. In the second part, presented in a companion paper, we define a local type inference system that allows the programmer to omit explicit instantiation annotations, and a type reconstruction system that allows the programmer to omit explicit type annotations. The work presented in the two articles provides the theoretical foundations and technical machinery needed to design and implement higher-order polymorphic functional languages for semi-structured data.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

Keywords Types, polymorphism, XML, intersection types

1. Introduction

The extensible markup language XML is a current standard format for exchanging structured data. Many recent XML processing languages, such as XDuce [14], ℂDuce [2], XQuery [3], OcamlDuce [11], XHaskell [17], XACT [15], are statically-typed functional languages. However, parametric polymorphism, an essential feature of such languages, is still missing, or when present it is in a limited form (no higher-order functions, no polymorphism for XML types, and so on). Polymorphism for XML has repeatedly been requested to and discussed in various working groups of standards (*eg*, RELAX NG [7]) and higher-order functions have been recently proposed in the W3C draft for XQuery 3.0 [10]. Despite all this interest, spurs, and motivations, a comprehensive polymorphic type system for XML was still missing for the simple reason that, until recently, it was deemed unfeasible. A major stumbling block to this research —*ie*, the definition of a subtyping relation for regular tree types with type variables— has been recently lifted by Castagna and Xu [6], who defined and studied a polymorphic subtyping relation for a type system with recursive, product, and arrow types and set-theoretic type connectives (union, intersection, and negation).

In this work we present the next logical step of that research, that is, the definition of a higher-order functional language that takes full advantage of the new capabilities of Castagna and Xu's system. In other words, we define and study a calculus with higher-order polymorphic functions and recursive types with union, intersection, and negation connectives. The approach is thus general and, as such, goes well beyond the simple application to XML processing languages. As a matter of facts, our motivating example developed all along this paper does not involve XML, but looks like a rather classic display of functional programming specimens:

```
map :: (α -> β) -> [α] -> [β]
map f l = case l of
          | [] -> []
          | (x : xs) -> (f x : map f xs)

even :: (Int -> Bool) ∧ ((α\Int) -> (α\Int))
even x = case x of
         | Int -> (x `mod` 2) == 0
         |   _ -> x
```

The first function is the classic `map` function defined in Haskell (we just used Greek letters to denote type variables). The second would be an Haskell function were it not for two oddities: its type contains type connectives (type intersection "∧" and type difference "\"); and the pattern in the `case` expression is a type, meaning that it matches all values returned by the matched expression that have that type. So what does the `even` function do? It checks whether its argument is an integer; if it is so it returns whether the integer is even or not, otherwise it returns its argument as it received it (although the definition of`even` may be considered weird, itis a perfect minimal example to illustrate all the aspects of our system).

The goal of this work is to define a calculus and a type system that can pass three tests. The first test is that it can define the two functions above. The second, harder, test is that the type system must be able to verify that these functions have the types declared in their signatures. That `map` has the declared type will come as no surprise (in practice, in the second part of this work we show that in the absence of a signature given by the programmer the system can reconstruct a type slightly more precise than this [5]). That `even` was given an intersection type means that it must have all the types that form the intersection. So it must be a function that when applied to an integer it returns a Boolean and that when applied to an argument of a type that does not contain any integer, it returns a result of the same type. In other terms, `even` is a polymorphic (dynamically bounded) overloaded function.

The third test, the hardest one, is that the type system must be able to *infer* the type of the partial application of `map` to `even`, and the inferred type must be equivalent to the following one[1]

```
map even :: ([Int] -> [Bool])   ∧
            ([α\Int] -> [α\Int]) ∧                    (1)
            ([α∨Int] -> [(α\Int)∨Bool])
```

[1] This type is redundant since the first type of the intersection is an instance (*eg*, for α=`Int`) of the third. We included it for the sake of the presentation.

POPL '14, January 22–24 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535840

since `map even` returns a function that when applied to a list of integers it returns a list of Booleans; when applied to a list that does not contain any integer then it returns a list of the same type (actually, the same list); and when it is applied to a list that may contain some integers (*eg*, a list of reals), then it returns a list of the same type, without the integers but with some Booleans instead (in the case of reals, a list with Booleans and reals that are not integers).

Technically speaking, the definition of such a calculus and its type system is difficult for two distinct reasons. First, for the reasons we explain in the next section, it demands to define an explicitly typed λ-calculus with intersection types, a task that, despite many attempts in the last 20 years, still lacked a satisfactory definition. Second, even if working with an explicitly typed setting may seem simpler, the system needs to solve "local type inference"[2], namely, the problem of checking whether the types of a function and of its argument can be made compatible and, if so, of inferring the type of their result as we did for (1). The difficulty, once more, mainly resides in the presence of the intersection types: a term can be given different types either by subsumption (the term is coerced into a super-type of its type) or by instantiation (the term is used as a particular instance of its polymorphic type) and it is typed by the intersection of all these types. Therefore, in this setting, the problem is not just to find a substitution that unifies the domain type of the function with the type of its argument but, rather, a *set* of substitutions that produce instances whose intersections are in the right subtyping relation: our `map even` example should already have given a rough idea of how difficult this is.

The presentation of our work is split in two parts, accordingly: in the first part (this paper) we show how to solve the problem of defining an explicitly-typed λ-calculus with intersection types and how to efficiently evaluate it; in the second part (the companion paper [5]) we will show how to solve the problem of "local type inference" for a calculus with intersection types. In the next section we outline the various problems we met (focusing on those that concern the part of the work presented in this paper) and how they were solved.

2. Problems and overview of the solution

The driver of this work is the definition of an XML processing functional language with high-order polymorphic functions, that is, in particular, a polymorphic version of the language $\mathbb{C}$Duce [2].

$\mathbb{C}$Duce in a nutshell. The essence of $\mathbb{C}$Duce is a λ-calculus with pairs, explicitly-typed recursive functions, and a type-case expression. Types can be recursively defined and include the arrow and product type *constructors* and the intersection, union, and negation type *connectives*. In summary, they are the regular trees coinductively generated by the following productions:

$$t ::= b \mid t \to t \mid t \times t \mid t \wedge t \mid t \vee t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \qquad (2)$$

where b ranges over basic types (*eg*, `Int`, `Bool`) and $\mathbb{0}$ and $\mathbb{1}$ respectively denote the empty (that types no value) and top (that types all values) types. We use possibly indexed meta-variables s and t to range over types. Coinduction accounts for recursive types. We use the standard convention that infix connectives have a priority higher than constructors and lower than prefix connectives.

From a strictly practical viewpoint, recursive types, products, and type connectives are used to encode regular tree types, which subsume existing XML schema/types while, for what concerns expressions, the type-case is an abstraction of $\mathbb{C}$Duce pattern matching (this uses regular expression patterns on types to define powerful and highly optimized capture primitives for XML data). We initially focus on the functional core and disregard products and recursive functions since the results presented here can be easily extended to them (we show it in the Appendix), though we will freely use them for our examples. So we initially consider the following "Core$\mathbb{C}$Duce" terms:

$$e ::= c \mid x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i} x.e \mid e \in t \,?\, e : e \qquad (3)$$

where c ranges over constants (*eg*, `true`, `false`, `1`, `2`, ...) which are values of basic types (we use b_c to denote the basic type of the constant c); x ranges over expression variables; $e \in t \,?\, e_1 : e_2$ denotes the type-case expression that evaluates either e_1 or e_2 according to whether the value returned by e (if any) is of type t or not; $\lambda^{\wedge_{i\in I} s_i \to t_i} x.e$ is a value of type $\wedge_{i\in I} s_i \to t_i$ that denotes the function of parameter x and body e.

In this work we show how to define the polymorphic extension of this calculus, which can then be easily extended to a full-fledged polymorphic functional language for processing XML documents. But before let us explain the two specificities of the terms in (3), namely, why a type-case expression is included and why we explicitly annotate whole λ-abstractions (with an intersection of arrow types) rather than just their parameters.

The reason of inclusion of a type-case is twofold. First, a natural application of intersection types is to type overloaded functions, and without a type-case only "coherent overloading" *à la* Forsythe [21] can be defined (which, for example, precludes in our setting the definition of a —non diverging— function of type, say, $(\texttt{Int}\to\texttt{Bool}) \wedge (\texttt{Bool}\to\texttt{Int})$). The second motivation derives from the way arrow types are interpreted in [6, 12]. In particular, for any types s_1, s_2, t_1, t_2 the following containment is, in general, *strict*:

$$s_1 \vee s_2 \to t_1 \wedge t_2 \quad \lneq \quad (s_1 \to t_1) \wedge (s_2 \to t_2) \qquad (4)$$

so there is a function in the type on the right that is not in the type of the left. Notice that from a typing viewpoint the functions on the left do not distinguish inputs of s_1 and s_2 types, while the ones on the right do. So the interpretation of types naturally induces the definition of functions that can distinguish inputs of two different types s_1 and s_2 whatever s_1 and s_2 are. Actually this second motivation is just a different facet of the full-fledged vs. only "coherent" overloading motivation, since the functions that are in the difference of the two types in (4) are also those that make the difference between coherent and non coherent overloading. Both arguments, thus, advocate for "real" overloaded functions, that execute different code according to the type of their input, whence the need of type-case. Therefore our terms include a type-case.

The need of explicitly typed functions is a direct consequence of the introduction of the type-case, because without explicit typing we can run into paradoxes such as the following recursively defined (constant) function

$$\mu f.\lambda x.f \in (\mathbb{1} \to \texttt{Int}) \,?\, \texttt{true} : \texttt{42} \qquad (5)$$

This function has type $\mathbb{1}\to\texttt{Int}$ if and only if it *does not* have type $\mathbb{1}\to\texttt{Int}$. In order to decide whether the function above is well-typed or not, we must explicitly give a type to it. For instance, the function in (5) is well-typed if it is explicitly assigned the type $\mathbb{1} \to \texttt{Int}\vee\texttt{Bool}$. This shows both that functions must be explicitly typed and that specifying not only the type of parameters but also the type of the result is strictly more expressive, as more terms can be typed. As a matter of fact, if we provide just the type of the parameter x (not used in the body), then there is no type (apart from the useless $\mathbb{0}$ type) that makes (5) typeable.

In summary, we need to define an explicitly typed language with intersection types. This is a difficult problem for which no full-fledged solution existed, yet: there exist only few intersection type systems with explicitly typed terms, and none of them is completely

[2] There are different definitions for *local type inference*. Here we use it with the meaning of finding the type of an expression in which not all type annotations are specified. This is the acceptation used in Scala where type parameters for polymorphic methods can be omitted. In our specific problem, we will omit —and, thus, infer— the annotations that specify how function and argument types can be made compatible.

satisfactory (see Section 7 on related work). To give an idea of why this is difficult, imagine we adopt for functions a Church-style notation as $\lambda x^t.e$ and consider the following "switch" function

$$\lambda x^t.\,(x{\in}\texttt{Int}\,\texttt{?}\,\texttt{true}:42) \tag{6}$$

that when applied to an `Int` returns `true` and returns `42` otherwise. Intuitively, we want to assign to this function the type $(\texttt{Int}{\to}\texttt{Bool})\wedge(\neg\texttt{Int}{\to}\texttt{Int})$, the type of a function that when applied to an `Int`, returns a `Bool`, and when applied to a value which is not an `Int`, returns an `Int`. For the sake of presentation, let us say that we are happy to deduce for the function above the less precise type $(\texttt{Int}{\to}\texttt{Bool})\wedge(\texttt{Bool}{\to}\texttt{Int})$ (which is a super-type of the former since if a function maps anything that is not an `Int` into an `Int`—it has type $\neg\texttt{Int}{\to}\texttt{Int}$—, then in particular it maps Booleans to integers —*ie*, it has also type $\texttt{Bool}{\to}\texttt{Int}$). The problem is to determine which type we should use for t in equation (6). If we use, say, $\texttt{Int}\vee\texttt{Bool}$, then under the hypothesis that $x:\texttt{Int}\vee\texttt{Bool}$ the type deduced for the body of the function is $\texttt{Int}\vee\texttt{Bool}$. So the best type we can give to the function in (6) is $\texttt{Int}\vee\texttt{Bool}\to\texttt{Int}\vee\texttt{Bool}$ which is far less precise than the sought intersection type, insofar as it does not make any distinction between arguments of type `Int` and those of type `Bool`.

The solution, which was introduced by ℂDuce, is to explicitly type —by an intersection type— whole λ-abstractions instead of just their parameters:

$$\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\texttt{Bool}\to\texttt{Int})}x\,.\,(x{\in}\texttt{Int}\,\texttt{?}\,\texttt{true}:42)$$

In doing so we also explicitly define the result type of functions which, as we have just seen, increases the expressiveness of the calculus. Thus the general form of λ-abstractions is, as stated by the grammar in (3), $\lambda^{\wedge_{i\in I}s_i\to t_i}x.e$. Such a term is well typed if for all $i\in I$ from the hypothesis that x has type s_i it is possible to deduce that e has type t_i. Unfortunately, with polymorphic types, this simple solution introduced by ℂDuce no longer suffices.

Polymorphic extension. The novelty of this work is to allow type variables (ranged over by lower-case Greek letters: $\alpha,\beta,\ldots$) to occur in the types in (2) and, thus, in the types labeling λ-abstractions in (3). It becomes thus possible to define the polymorphic identity function as $\lambda^{\alpha\to\alpha}x.x$, while classic "auto-application" term can be written as $\lambda^{((\alpha\to\beta)\wedge\alpha)\to\beta}x.xx$. The intended meaning of using a type variable, such as α, is that a (well-typed) λ-abstraction not only has the type specified in its label (and by subsumption all its super-types) but also all the types obtained by instantiating the type variables occurring in the label. So $\lambda^{\alpha\to\alpha}x.x$ has not only type $\alpha\to\alpha$ but also, for example, by subsumption the types $\mathbb{0}\to\mathbb{1}$ (the type of all functions, which is a super-type of $\alpha\to\alpha$) and $\neg\texttt{Int}$ (since every well-typed λ-abstraction is not an integer, then $\neg\texttt{Int}$ contains —*ie*, is a super-type of— all function types), and by instantiation the types $\texttt{Int}\to\texttt{Int}$, $\texttt{Bool}\to\texttt{Bool}$, etc.

The use of instantiation in combination with intersection types has nasty consequences, for if a term has two distinct types, then it has also their intersection type (*eg*, $\lambda^{\alpha\to\alpha}x.x$ has type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})\wedge\neg\texttt{Int}$). In the monomorphic case a term can have distinct types only by subsumption and, thus, intersection types are transparently assigned to terms via subsumption. But in the polymorphic case this is no longer possible: a term can be typed by the intersection of two distinct instances of its polymorphic type which, in general, are not in any subtyping relation with the latter: for instance, $\alpha\to\alpha$ is neither a subtype of $\texttt{Int}\to\texttt{Int}$ nor vice versa, since the subtyping relation must hold for *all possible* instantiations of α and there are infinitely many instances of $\alpha\to\alpha$ that are neither a subtype nor a super-type of $\texttt{Int}\to\texttt{Int}$.

Explicit instantiation. Concretely, if we want to apply the polymorphic identity $\lambda^{\alpha\to\alpha}x.x$ to, say, `42`, then the particular instance obtained by the type-substitution $\{\texttt{Int}/\alpha\}$ (denoting the replacement of every occurrence of α by `Int`) must be used, that is $(\lambda^{\texttt{Int}\to\texttt{Int}}x.x)42$. We have thus to *relabel* the type decorations of λ-abstractions before applying them. In implicitly typed languages, such as ML, the relabeling is meaningless (no type decoration is used in terms) while in their explicitly-typed counterparts relabeling can be seen as a logically meaningful but computationally useless operation, insofar as execution takes place on type erasures (*ie*, the terms obtained by erasing all type decorations). In the presence of type-case expressions, however, relabeling is necessary since the label of a λ-abstraction determines its type: testing whether an expression has type, say, $\texttt{Int}\to\texttt{Int}$ should succeed for the application of $\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x$ to `42` and fail for its application to `true`. In practice, we have that

$$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)42{\in}\texttt{Int}\to\texttt{Int}\,\texttt{?}\,0:1$$

must reduce to $\lambda^{\texttt{Int}\to\texttt{Int}}y.42{\in}\texttt{Int}\to\texttt{Int}\,\texttt{?}\,0:1$ and thus to `0`, while

$$(\lambda^{\alpha\to\alpha\to\alpha}x.\lambda^{\alpha\to\alpha}y.x)\texttt{true}{\in}\texttt{Int}\to\texttt{Int}\,\texttt{?}\,0:1$$

must reduce to $\lambda^{\texttt{Bool}\to\texttt{Bool}}y.\texttt{true}{\in}\texttt{Int}\to\texttt{Int}\,\texttt{?}\,0:1$ and thus to `1`. This means that, in Reynolds' terminology, our terms have an *intrinsic* meaning [22], that is to say, the semantics of a term depends on its typing.

If we need to relabel some function, then it may be necessary to relabel also its body as witnessed by the following "daffy" —though well-typed— definition of the identity function:

$$(\lambda^{\alpha\to\alpha}x.(\lambda^{\alpha\to\alpha}y.x)x) \tag{7}$$

If we want to apply this function to, say, `3`, then we have first to relabel it by applying the substitution $\{\texttt{Int}/\alpha\}$. However, applying the relabeling only to the outer "λ" does not suffice since the application of (7) to `3` reduces to $(\lambda^{\alpha\to\alpha}y.3)3$ which is not well-typed (it is not possible to deduce the type $\alpha\to\alpha$ for $\lambda^{\alpha\to\alpha}y.3$, which is the constant function that always returns `3`) although it is the reductum of a well-typed application.[3]

The solution is to apply the relabeling also to the body of the function. Here what "to relabel the body" means is straightforward: apply the same type-substitution $\{\texttt{Int}/\alpha\}$ to the body. This yields a reductum $(\lambda^{\texttt{Int}\to\texttt{Int}}y.3)3$ which is well typed. In general, however, the way to perform a relabeling of the body of a function is not so straightforward and clearly defined, since two different problems may arise: (i) it may be necessary to apply more than a single type-substitution and (ii) the relabeling of the body may depend on the dynamic type of the actual argument of the function (both problems are better known as —or are instances of— the problem of determining expansions for intersection type systems [8]). Next, we discuss each problem in detail.

Multiple substitutions. First of all, notice that we may need to relabel/instantiate functions not only when they are applied but also when they are used as arguments. For instance, consider a function that expects arguments of type $\texttt{Int}\to\texttt{Int}$. It is clear that we can apply it to the identity function $\lambda^{\alpha\to\alpha}x.x$, since the identity function *has* type $\texttt{Int}\to\texttt{Int}$ (feed it by an integer and it will return an integer). Before, though, we have to relabel the latter by the substitution $\{\texttt{Int}/\alpha\}$ yielding $\lambda^{\texttt{Int}\to\texttt{Int}}x.x$. As the identity $\lambda^{\alpha\to\alpha}x.x$ has type $\texttt{Int}\to\texttt{Int}$, so it has type $\texttt{Bool}\to\texttt{Bool}$ and, therefore, the intersection of the two: $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$. So we can apply a function that expects an argument of this intersection type to our identity function. The problem is now how to relabel $\lambda^{\alpha\to\alpha}x.x$. Intuitively, we have to apply two distinct type-substitutions $\{\texttt{Int}/\alpha\}$ and $\{\texttt{Bool}/\alpha\}$ to the label of the λ-abstraction

[3] By convention a type variable is introduced by the outermost λ in which it occurs and this λ implicitly binds all inner occurrences of the variable. For instance, all the α's in the term (7) are the same while in a term such as $(\lambda^{\alpha\to\alpha}x.x)(\lambda^{\alpha\to\alpha}x.x)$ the variables in the function are distinct from those in its argument and, thus, can be α-converted separately, as $(\lambda^{\gamma\to\gamma}x.x)(\lambda^{\delta\to\delta}x.x)$.

and replace it by the intersection of the two instances. This corresponds to relabel the polymorphic identity from $\lambda^{\alpha\to\alpha}x.x$ into $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.x$. This is the solution adopted by this work, where we manipulate *sets of type-substitutions* —delimited by square brackets. The application of such a set (*eg*, in the previous example $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$) to a type t returns the intersection of all types obtained by applying each substitution in the set to t (*eg*, in the example $t\{\texttt{Int}/\alpha\}\wedge t\{\texttt{Bool}/\alpha\}$). Thus the first problem has an easy solution.

Relabeling of function bodies. The second problem is much harder and concerns the relabeling of the body of a function. While the naive solution consisting of propagating the application of type-substitutions to the bodies of functions works for single type-substitutions, in general, it fails for *sets* of type-substitutions. This can be seen by considering the relabeling via the set of type-substitutions $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$ of the daffy function in (7). If we apply the naive solution, this yields

$$(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}x.(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x) \qquad (8)$$

which is not well typed. That this term is not well typed is clear if we try applying it to, say, `3`: the application of a function of type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ to an `Int` should have type `Int`, but here it reduces to $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.3)3$, and there is no way to deduce the intersection type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ for the constant function $\lambda y.3$. But we can also directly verify that it is not well typed, by trying typing the function in (8). This corresponds to prove that under the hypothesis x : `Int` the term $(\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x)x$ has type `Int`, and that under the hypothesis x : `Bool` this same term has type `Bool`. Both checks fail because, in both cases, $\lambda^{(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})}y.x$ is ill-typed (it neither has type $\texttt{Int}\to\texttt{Int}$ when x:`Bool`, nor has it type $\texttt{Bool}\to\texttt{Bool}$ when x:`Int`). This example shows that in order to ensure that relabeling yields well-typed terms, the relabeling of the body *must change* according to the type of the value the parameter x is bound to. More precisely, $(\lambda^{\alpha\to\alpha}y.x)$ should be relabeled as $\lambda^{\texttt{Int}\to\texttt{Int}}y.x$ when x is of type `Int`, and as $\lambda^{\texttt{Bool}\to\texttt{Bool}}y.x$ when x is of type `Bool`. An example of this same problem less artificial than our daffy function is given by the classic `apply` function $\lambda f.\lambda x.fx$ which, with our polymorphic type annotations, is written as:

$$\lambda^{(\alpha\to\beta)\to\alpha\to\beta}f.\lambda^{\alpha\to\beta}x.fx \qquad (9)$$

The `apply` function in (9) has type $(\texttt{Int}\to\texttt{Int})\to\texttt{Int}\to\texttt{Int}$, obtained by instantiating its type annotation by the substitution $\{\texttt{Int}/\alpha,\texttt{Int}/\beta\}$, as well as type $(\texttt{Bool}\to\texttt{Bool})\to\texttt{Bool}\to\texttt{Bool}$, obtained by the substitution $\{\texttt{Bool}/\alpha,\texttt{Bool}/\beta\}$. If we want to feed this function to another function that expects arguments whose type is the intersections of these two types, then we have to relabel it by using the set of type-substitutions $[\{\texttt{Int}/\alpha,\texttt{Int}/\beta\},\{\texttt{Bool}/\alpha,\texttt{Bool}/\beta\}]$. But, once more, it is easy to verify that the naive solution that consists in propagating the application of the set of type-substitutions down to the body of the function yields an ill-typed expression.

This second problem is the showstopper for the definition of an explicitly typed λ-calculus with intersection types. Most of the solutions found in the literature [4, 16, 23, 26] rely on the duplication of lambda terms and/or typing derivations, while other calculi such as [27] that aim at avoiding such duplication obtain it by adding new expressions and new syntax for types (see related work in Section 7); but none of them is able to produce an explicitly-typed λ-calculus with intersection types, as we do, by just adding annotations to λ-abstractions.

Our solution. Here we introduce a new technique that consists in performing a "lazy" relabeling of the bodies. This is obtained by decorating λ-abstractions by (sets of) type-substitutions. For example, in order to pass our daffy identity function (7) to a function that expects arguments of type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$, we first "lazily" relabel it as follows:

$$(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x). \qquad (10)$$

The new annotation in the outer "λ" indicates that the function must be relabeled and, therefore, that we are using the particular instance whose type is the one in the interface (*ie*, $\alpha\to\alpha$) to which we apply the set of type-substitutions. The relabeling will be actually propagated to the body of the function at the moment of the reduction, only if and when the function is applied (relabeling is thus lazy). However, the new annotation is statically used by the type system to check soundness. Notice that, unlike existing solutions, we preserve the structure of λ-terms (at the expenses of some extra annotation that is propagated during the reduction) which is of the uttermost importance in a language-oriented study.

In this paper we focus on the study of the calculus with these "lazy" type-substitutions annotations. We temporarily avoid the problem of local type inference by defining a *calculus with explicit sets of type substitutions*: expressions will be explicitly annotated with appropriate sets of type-substitutions.

Polymorphic $\mathbb{C}$Duce. From a practical point of view, however, it is important to stress that, at the end, these annotations will be invisible to the programmer and, as we show in the second part presented in the companion paper [5], all the necessary type-substitutions will be inferred statically. In practice, the programmer will program in the language defined by grammar (3), but where the types that annotate λ's may contain type variables, that is, the polymorphic version of $\mathbb{C}$Duce. The problem of inferring explicit sets of type-substitutions to annotate the polymorphic version of the expressions in (3) is the topic of the second part of this work presented in the companion paper [5]. For the time being, simply notice that the language defined by (3) and extended with type variables passes our first test inasmuch as the `even` function can be defined as follows (where $s\backslash t$ is syntactic sugar for $s\wedge\neg t$):

$$\lambda^{(\texttt{Int}\to\texttt{Bool})\wedge(\alpha\backslash\texttt{Int}\to\alpha\backslash\texttt{Int})}x\,.\,x{\in}\texttt{Int}\,?\,(x \bmod 2)=0:x \qquad (11)$$

while —with the products and recursive functions described in the Appendix— `map` is defined as (see also discussion in Appendix E)

$$\mu m^{(\alpha\to\beta)\to[\alpha]\to[\beta]}f = \lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell{\in}\texttt{nil}\,?\,\texttt{nil}:(f(\pi_1\ell),mf(\pi_2\ell)) \qquad (12)$$

where the type `nil` tested in the type-case denotes the singleton type that contains just the constant `nil`, and $[\alpha]$ denotes the regular type that is the (least) solution of $X=(\alpha,X)\vee\texttt{nil}$.

When fed by any expression of this language, the type inference system defined in the companion paper [5] will infer sets of type-substitutions and insert them into the expression to make it well typed (if possible, of course). For example, for the application (of the terms defining) `map` to `even`, the inference system of the companion paper [5] infers the following set of type-substitutions $[\{(\alpha\backslash\texttt{Int})/\alpha,(\alpha\backslash\texttt{Int})/\beta\},\{\alpha\vee\texttt{Int}/\alpha,(\alpha\backslash\texttt{Int})\vee\texttt{Bool}/\beta\}]$ and textually inserts it between the two terms (so that the type-substitutions apply to the type variables of `map`) yielding the typing in (1). Finally, as we explain in Section 5.3 later on, the compiler will compile the expression into an expression of an intermediate language that can be evaluated as efficiently as the monomorphic calculus.

Outline. The rest of the presentation proceeds as follows. In Section 3 we define and study our calculus with explicit type-substitutions: we define its syntax, its operational semantics, and its type system; we prove that the type system is sound and subsumes classic intersection type systems. In Section 4 we define an algorithm for type inference and prove that it is sound, complete, and terminating. In Section 5 we show that the addition of type-substitutions has in practice no impact on the efficiency of the evaluation since the calculus can be compiled into an intermediate language that executes as efficiently as monomorphic $\mathbb{C}$Duce.

Section 7 presents related work and in Section 8 we conclude our presentation.

In the rest of the presentation we will focus on the intuition and try to avoid as many technical details as possible. We dot the i's and cross the t's in the Appendix, where all formal definitions and complete proofs of properties can be found (*n.b.*: references in the text starting by capital letters —*eg*, Definition A.7— refer to this appendix). All these as well as other details can also be found in the third author's PhD thesis manuscript [28].

Contributions. The overall contribution of our work is the definition of a statically-typed core language with (i) polymorphic higher-order functions for a type system with recursive types and union, intersection, and negation type connectives, (ii) an efficient evaluation model, (iii) local type inference for application, and (iv) a limited form of type reconstruction.

The main contribution of this first part of the work is the definition of an *explicitly-typed* λ-calculus (actually, a family of calculi) with intersection (and union and negation) types and of its efficient evaluation via the compilation into an intermediate language. From a syntactic viewpoint our solution is a minimal extension since it just requires to add annotations to λ-abstractions of the untyped λ-calculus (*cf.* Section 3.5). Although this problem has been studied for over 20 years, no existing solution proposes such a minimal extension, which is of paramount importance in a programming language-oriented study (see related works in Section 7).

The technical contributions are the definition of an explicitly typed calculus with intersection types; the proof that it subsumes existing intersection type systems; the soundness of its type system, the definition of a sound, complete and terminating algorithm for type inference (which as byproduct yields an intersection type proof system satisfying the Curry-Howard isomorphism); the definition of a compilation technique into an intermediate language that can be evaluated as efficiently as the monomorphic one; its extension to the so called `let`-polymorphism and the proof of the adequacy of the compilation. Local type inference for application and type reconstruction are studied in the second part of this work presented in the companion paper [5].

3. A calculus with explicit type-substitutions

The types of the calculus are those in the grammar (2) to which we add type variables (ranged over by α) and, for the sake of presentation, stripped of product types. In summary, types are the regular trees coinductively generated by

$$t ::= \alpha \mid b \mid t \to t \mid t \wedge t \mid t \vee t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \qquad (13)$$

and such that every infinite branch contains infinitely many occurrences of type constructors. We use $\mathscr{T}$ to denote the set of all types. The condition on infinite branches bars out ill-formed types such as $t = t \vee t$ (which does not carry any information about the set denoted by the type) or $t = \neg t$ (which cannot represent any set). It also ensures that the binary relation $\rhd \subseteq \mathscr{T}^2$ defined by $t_1 \vee t_2 \rhd t_i$, $t_1 \wedge t_2 \rhd t_i$, $\neg t \rhd t$ is Noetherian (that is, strongly normalizing). This gives an induction principle on $\mathscr{T}$ that we will use without any further explicit reference to the relation. We use $\mathsf{var}(t)$ to denote the *set of type variables* occurring in a type t (see Definition A.2). A type t is said to be *ground* or *closed* if and only if $\mathsf{var}(t)$ is empty.

The subtyping relation for the types in $\mathscr{T}$ is the one defined by Castagna and Xu [6]. For this work it suffices to consider that ground types are interpreted as sets of *values* (*n.b.*, just values, not expressions) that have that type and subtyping is set containment (*ie*, a ground type s is a subtype of a ground type t if and only if t contains all the values of type s). In particular, $s \to t$ contains all λ-abstractions that when applied to a value of type s, if they return a result, then this result is of type t (so $\mathbb{0} \to \mathbb{1}$ is the set of all functions and $\mathbb{1} \to \mathbb{0}$ is the set of functions that diverge on all arguments). Type connectives (union, intersection, negation) are interpreted as the corresponding set-theoretic operators and subtyping is set containment. For what concerns non-ground types (*ie*, types with variables occurring in them) all the reader needs to know for this work is that the subtyping relation of Castagna and Xu is preserved by type-substitutions. Namely, if $s \leq t$, then $s\sigma \leq t\sigma$ for every type-substitution σ (the converse does not hold in general, while it holds for *semantic* type-substitutions in convex models: see [6]). Two types are equivalent if they denote the same set of values, that is, if they are subtype one of each other (type equivalence is denoted by $\simeq$). An important property of this system we will often use is that every type is equivalent to (and can be effectively transformed into) a type in *disjunctive normal form*, that is, a union of *uniform* intersections of literals. A literal is either an arrow, or a basic type, or a type variable, or their negations. An intersection is uniform if all the literals have the same constructor, that is, either it is an intersection of arrows, type variables, and their negations or it is an intersection of basic types, type variables, and their negations. In summary, a disjunctive normal form is a union of summands whose form is either

$$\bigwedge_{p\in P} b_p \wedge \bigwedge_{n\in N} \neg b_n \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r \qquad (14)$$

or

$$\bigwedge_{p\in P} (s_p \to t_p) \wedge \bigwedge_{n\in N} \neg(s_n \to t_n) \wedge \bigwedge_{q\in P'} \alpha_q \wedge \bigwedge_{r\in N'} \neg\alpha_r \qquad (15)$$

When either P' or N' is not empty, we call the variables α_q's and α_r's the *top-level variables* of the normal form.

3.1 Expressions

Expressions are derived from those of Core$\mathbb{C}$Duce (with type variables in types) with the addition that sets of explicit type-substitutions (ranged over by $[\sigma_j]_{j\in J}$) may be applied to terms and decorate λ-abstractions

$$e ::= c \mid x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e\in t\,?\,e : e \mid e[\sigma_j]_{j\in J} \qquad (16)$$

and with the restriction that the type tested in type-case expressions is closed. Henceforth, given a λ-abstraction $\lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e$ we call the type $\bigwedge_{i\in I} s_i \to t_i$ the *interface* of the function and the set of type-substitutions $[\sigma_j]_{j\in J}$ the *decoration* of the function. We write $\lambda^{\wedge_{i\in I} t_i\to s_i} x.e$ for short when the decoration is a singleton containing the empty substitution. Let e be an expression. We use $\mathsf{fv}(e)$ and $\mathsf{bv}(e)$ respectively to denote the sets of *free expression variables* and *bound expressions variables* of the expression e; we use $\mathsf{tv}(e)$ to denote the set of *type variables* occurring in e (see Definition A.8).

As customary, we assume bound expression variables to be pairwise distinct and distinct from any free expression variable occurring in the expressions under consideration. We equate expressions up to the α-renaming of their bound expression variables. In particular, when substituting an expression e for a variable y in an expression e' (see Definition A.10), we assume that the bound variables of e' are distinct from the bound and free variables of e, to avoid unwanted captures. For example, $(\lambda^{\alpha\to\alpha} x.x)y$ is α-equivalent to $(\lambda^{\alpha\to\alpha} z.z)y$.

The situation is a bit more complex for type variables, as we do not have an explicit binder for them. Intuitively, a type variable can be α-converted if it is a *polymorphic* one, that is, if it can be instantiated. For example, $(\lambda^{\alpha\to\alpha} x.x)y$ is α-equivalent to $(\lambda^{\beta\to\beta} x.x)y$, and $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} x.x)y$ is α-equivalent to $(\lambda^{\beta\to\beta}_{[\{\mathtt{Int}/\beta\}]} x.x)y$. Polymorphic variables can be bound by interfaces, but also by decorations: for example, in $\lambda^{\beta\to\beta}_{[\{\alpha/\beta\}]} x.(\lambda^{\alpha\to\alpha} y.y)x$, the α occurring in the interface of the inner abstraction is "bound" by the decoration $[\{\alpha/\beta\}]$, and the whole expression is α-equivalent to

$$\text{(subsum)}\ \frac{\Delta \wr \Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Delta \wr \Gamma \vdash e : t_2} \qquad \text{(appl)}\ \frac{\Delta \wr \Gamma \vdash e_1 : t_1 \to t_2 \quad \Delta \wr \Gamma \vdash e_2 : t_1}{\Delta \wr \Gamma \vdash e_1 e_2 : t_2}$$

$$\text{(abstr)}\ \frac{\Delta \cup \mathsf{var}(\wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \wr \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j}{\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j} \begin{array}{l} i \in I \\ j \in J \end{array}$$

$$\text{(var)}\ \frac{}{\Delta \wr \Gamma \vdash x : \Gamma(x)} \qquad \text{(case)}\ \frac{\Delta \wr \Gamma \vdash e : t' \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \Delta \wr \Gamma \vdash e_1 : s \\ t' \not\leq t & \Rightarrow \quad \Delta \wr \Gamma \vdash e_2 : s \end{cases}}{\Delta \wr \Gamma \vdash (e \in t\,?\,e_1 : e_2) : s}$$

$$\text{(inst)}\ \frac{\Delta \wr \Gamma \vdash e : t \quad \sigma \sharp \Delta}{\Delta \wr \Gamma \vdash e[\sigma] : t\sigma} \qquad \text{(inter)}\ \frac{\forall j \in J.\ \Delta \wr \Gamma \vdash e[\sigma_j] : t_j}{\Delta \wr \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ |J| > 1$$

Figure 1. Static semantics

$(\lambda^{\beta\to\beta}_{[\{\gamma/\beta\}]} x.(\lambda^{\gamma\to\gamma} y.y)x)$. If a type variable is bound by an outer abstraction, it cannot be instantiated; such a variable is called *monomorphic*. For example, the following expression

$$\lambda^{(\alpha\to\alpha)\to(\alpha\to\alpha)} y.((\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/\alpha\}]y)$$

is not sound (*ie*, it cannot be typed), because α is bound at the level of the outer abstraction, not at level of the inner one. Consequently, in this expression, α is monomorphic for the inner abstraction, but polymorphic for the outer one (strictly speaking, thus, the monomorphic and polymorphic adjectives apply to occurrences of variables rather than variables themselves). Monomorphic type variables cannot be α-converted: $\lambda^{(\alpha\to\alpha)\to(\alpha\to\alpha)} y.(\lambda^{\alpha\to\alpha} x.x)y$ is *not* α-equivalent to $\lambda^{(\alpha\to\alpha)\to(\alpha\to\alpha)} y.(\lambda^{\beta\to\beta} x.x)y$ (but it is α-equivalent to $\lambda^{(\beta\to\beta)\to(\beta\to\beta)} y.(\lambda^{\beta\to\beta} x.x)y$). Note that the scope of polymorphic variables may include some type-substitutions $[\sigma_i]_{i\in I}$: for example, $((\lambda^{\alpha\to\alpha} x.x)y)[\mathtt{Int}/\alpha]$ is α-equivalent to $((\lambda^{\beta\to\beta} x.x)y)[\mathtt{Int}/\beta]$. Finally, we have to be careful when performing expression substitutions and type-substitutions to avoid clashes of polymorphic variable namespaces. For example, substituting $\lambda^{\alpha\to\alpha} z.z$ for y in $\lambda^{\alpha\to\alpha} x.x\ y$ would lead to an unwanted capture of α (assuming α is polymorphic, that is, not bound by a λ-abstraction placed above these two expressions), so we have to α-convert one of them, so that the result of the substitution is, for instance, $\lambda^{\alpha\to\alpha} x.x\ (\lambda^{\beta\to\beta} z.z)$.

To resume, we assume polymorphic variables to be pairwise distinct and distinct from any monomorphic variable in the expressions under consideration. We equate expressions up to α-renaming of their polymorphic variables. In particular, when substituting an expression e' for a variable x in an expression e, we suppose the polymorphic type variables of e to be distinct from the monomorphic and polymorphic type variables of e' thus avoiding unwanted captures. Detailed definitions are given in Appendix A.2.

In order to define both static and dynamic semantics for the expressions above, we need to define the *relabeling* operation "@" which takes an expression e and a set of type-substitutions $[\sigma_j]_{j\in J}$ and pushes $[\sigma_j]_{j\in J}$ to all outermost λ-abstractions occurring in e (and collects and composes with the sets of type-substitutions it meets). Precisely, $e@[\sigma_j]_{j\in J}$ is defined for λ-abstractions and (inductively) for applications of type-substitutions as:

$$\begin{array}{rcl} (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e \\ (e[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J} & \stackrel{\text{def}}{=} & e@([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) \end{array}$$

where $\circ$ denotes the pairwise composition of all substitutions of the two sets (see Definition 3.1). It erases the set of type-substitutions when e is a variable and it is homomorphically applied on the remaining expressions (see Definition A.11).

Formally the composition of two sets of type-substitutions is defined as follows:

Definition 3.1. *Given two sets of type-substitutions* $[\sigma_i]_{i\in I}$ *and* $[\sigma_j]_{j\in J}$*, we define their* composition *as*

$$[\sigma_i]_{i\in I} \circ [\sigma_j]_{j\in J} = [\sigma_i \circ \sigma_j]_{i\in I, j\in J}$$

where

$$\sigma_i \circ \sigma_j(\alpha) = \begin{cases} (\sigma_j(\alpha))\sigma_i & \text{if } \alpha \in \mathsf{dom}(\sigma_j) \\ \sigma_i(\alpha) & \text{if } \alpha \in \mathsf{dom}(\sigma_i) \setminus \mathsf{dom}(\sigma_j) \\ \alpha & \text{otherwise} \end{cases}$$

Next, we formally define the relabeling of an expression e with a set of type substitutions $[\sigma_j]_{j\in J}$, which consists in propagating the σ_j to the λ-abstractions in e if needed. We suppose that the polymorphic type variables in e are distinct from the type variables in the range of σ_j (this is always possible by using α-conversion).

Definition 3.2 (Relabeling). *Given an expression* e *and a set of type-substitutions* $[\sigma_j]_{j\in J}$*, we define the* relabeling *of* e *with* $[\sigma_j]_{j\in J}$*, written* $e@[\sigma_j]_{j\in J}$*, as* e *if* $\mathsf{tv}(e) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$*, and otherwise as follows:*

$$\begin{array}{rcl} (e_1\ e_2)@[\sigma_j]_{j\in J} & = & (e_1@[\sigma_j]_{j\in J})\ (e_2@[\sigma_j]_{j\in J}) \\ (e \in t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & = & e@[\sigma_j]_{j\in J} \in t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\ (e[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J} & = & e@([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) \\ (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & = & \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e \end{array}$$

The substitutions are not propagated if they do not affect the variables of e (*ie*, if $\mathsf{tv}(e) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$). In particular, constants and variables are left unchanged, as they do not contain any type variable.

3.2 Operational semantics

The dynamic semantics is given by the following three notions of reduction (where v ranges over *values*, that is, constants and λ-abstractions), applied by a leftmost-outermost strategy:

$$e[\sigma_j]_{j\in J} \quad \rightsquigarrow \quad e@[\sigma_j]_{j\in J} \tag{17}$$

$$(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e)v \quad \rightsquigarrow \quad (e@[\sigma_j]_{j\in P})\{v/x\} \tag{18}$$

$$v \in t\,?\,e_1 : e_2 \quad \rightsquigarrow \quad \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases} \tag{19}$$

where in (18) we have $P \stackrel{\text{def}}{=} \{j \in J \mid \exists i \in I, \vdash v : t_i\sigma_j\}$.

The first rule (17) performs relabeling, that is, it propagates the sets of type-substitutions down into the decorations of the outermost λ-abstractions. The second rule (18) states the semantics of applications: this is standard call-by-value β-reduction, with the difference that the substitution of the argument for the parameter is performed on the relabeled body of the function. Notice that relabeling depends on the type of the argument and keeps only those substitutions that make the type of the argument v match (at least one of) the input types defined in the interface of the function (*ie*, the set P which contains all substitutions σ_j such that the argument v has type $t_i\sigma_j$ for some i in I: the type system will ensure that P is never empty). For instance, take the daffy identity function, instantiate it as in (10) by both `Int` and `Bool`, and apply it to `42` —*ie*, $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}]} x.(\lambda^{\alpha\to\alpha} y.x)x)\mathtt{42}$—, then it reduces to $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} y.\mathtt{42})\mathtt{42}$, (which is observationally equivalent to $(\lambda^{\mathtt{Int}\to\mathtt{Int}} y.\mathtt{42})\mathtt{42}$) since the reduction discards the $\{\mathtt{Bool}/\alpha\}$

substitution. Finally, the third rule (19) checks whether the value returned by the expression in the type-case matches the specified type and selects the branch accordingly.

The reader may think that defining a functional language in which each β-reduction must perform an involved relabeling operation, theoretically interesting though it may be, will result in practice too costly and therefore unrealistic. This is not so. In Section 5 we show that this reduction can be implemented as efficiently as in $\mathbb{C}$Duce. By a smart definition of closures it is possible to compute relabeling in a lazy way and materialize it only in a very specific case for the reduction of the type-case (*ie*, to perform a type-case reduction (19) where the value v is a function whose interface contains monomorphic type variables and it is the result of the partial application of a polymorphic function) while all other reductions for applications can be implemented as plain classic β-reduction. For instance, to evaluate the expressions $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\},\{\mathtt{Bool}/\alpha\}}x.(\lambda^{\alpha\to\alpha}y.x)x)\mathtt{42}$ above, we can completely disregard all type annotations and decorations and perform a couple of standard β reductions that yield the result `42`.

3.3 Type system

As expected in a calculus with a type-case expression, the dynamic semantics depends on the static semantics —precisely, on the typing of values. The static semantics of our calculus is defined in Figure 1. The judgments are of the form $\Delta\,;\Gamma \vdash e : t$, where e is an expression, t a type, Γ a type environment (*ie*, a finite mapping from expression variables to types), and Δ a finite set of type variables. The latter is the set of all *monomorphic type variables*, that is, the variables that occur in the type of some outer λ-abstraction and, as such, cannot be instantiated; it must contain all the type variables occurring in Γ.

The rules for application and subsumption are standard. In the latter, the subtyping relation is the one defined in [6]. We just omitted the rule for constants (which states that c has type b_c).

The rule for abstractions applies each substitution specified in the decoration to each arrow type in the interface, adds all the variables occurring in these types to the set of monomorphic type variables Δ, and checks whether the function has all the resulting types. Namely, it checks that for every possible input type, the (relabeled) body has the corresponding output type. To that end, it applies each substitution σ_j in the decoration to each input type t_i of the interface and checks that, under the hypothesis that x has type $t_i\sigma_j$, the function body relabeled with the substitution σ_j at issue has type $s_i\sigma_j$ (notice that all these checks are performed under the same updated set of monomorphic type variables, that is, $\Delta \cup \mathsf{var}(\wedge_{i\in I,j\in J}t_i\sigma_j\to s_i\sigma_j)$). If the test succeeds, then the rule infers for the function the type obtained by applying the set of substitutions of the decoration to the type of the interface. For example, in the case of the instance of the daffy identity function given in (10), the Δ is always empty and the rule checks whether under the hypothesis $x : \alpha\{\mathtt{Int}/\alpha\}$ (*ie*, $x : \mathtt{Int}$), it is possible to deduce that $(\lambda^{\alpha\to\alpha}y.x)x@[\{\mathtt{Int}/\alpha\}]$ has type $\alpha\{\mathtt{Int}/\alpha\}$ (*ie*, that $(\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x)x : \mathtt{Int}$), and similarly for the substitution $\{\mathtt{Bool}/\alpha\}$. The type deduced for the function is then $(\mathtt{Int}\to\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool})$. The relabeling of the body in the premises of the rule (*abstr*) is a key mechanism of the type system: had we used $e[\sigma_j]$ instead of $e@[\sigma_j]$ in the premises of the (*abstr*) rule, the expression (10) could not be typed. The reason is that $e[\sigma_j]$ is more demanding on typing than $e@[\sigma_j]$, since the well typing of e is necessary to the well-typing of the former but not to that of the latter. Indeed while under the hypothesis $x : \mathtt{Int}$ we just showed that $((\lambda^{\alpha\to\alpha}y.x)x)@[\{\mathtt{Int}/\alpha\}]$ —*ie*, $((\lambda^{\mathtt{Int}\to\mathtt{Int}}y.x)x)$— is well-typed, the term $((\lambda^{\alpha\to\alpha}y.x)x)[\{\mathtt{Int}/\alpha\}]$ is not, for $(\lambda^{\alpha\to\alpha}y.x)$ does not have type $\alpha\to\alpha$. The rule for abstractions also justifies the need for an explicit set Δ for monomorphic type variables while, for instance, in ML it suffices to consider monomorphic type variables that occur in the image of Γ [20]: when checking an arrow of the interface of a function, the variables occurring in the other arrows must be considered monomorphic, too.

To type the applications of a set of type-substitutions to an expression, two different rules are used according to whether the set contains one or more than one substitution. When a single substitution is specified, the rule (*inst*) instantiates the type according to the specified substitution, provided that σ does not substitute variables in Δ (*ie*, $\mathsf{dom}(\sigma)\cap\Delta = \varnothing$, noted $\sigma\sharp\Delta$). This condition is necessary to the soundness of the system. Without it an expression such as $\lambda^{(\alpha\to\beta)}x.x[\{\beta/\alpha\}]$ could be typed as follows:

$$\dfrac{\dfrac{\overline{\{\alpha,\beta\}\,;\{(x:\alpha)\}\vdash x:\alpha}}{\{\alpha,\beta\}\,;\{(x:\alpha)\}\vdash x[\{\beta/\alpha\}]:\beta}}{\vdash \lambda^{(\alpha\to\beta)}x.x[\{\beta/\alpha\}]:\alpha\to\beta}$$

which is unsound (by applying the above functions to any value it is possible to create polymorphic values of type β, for every β). If more than one substitution is specified, then the rule (*inter*) composes them by an intersection. Notice that the type system composes by intersection only different types of a same term obtained by instantiation. This is not restrictive since different types obtained by subsumption can be composed by intersection by applying just subsumption (see Lemma B.2).

Finally, the (*case*) rule first infers the type t' of the expression whose type is tested. Then the type of each branch e_i is checked only if there is a chance that the branch can be selected. Here the use of "$\not\leq$" is subtle but crucial (it allows us to existentially quantify over type-substitutions). The branch, say, e_1 can be selected (and therefore its well-typedness must be checked) only if e can return a value that is in t. But in order to cover all possible cases we must also consider the case in which the type of e is instantiated as a consequence of an outer application. A typical usage pattern (followed also by our `even` function) is $\lambda^{\alpha\to\cdots}x.\ x{\in}\mathtt{Int}\,?\,e_1 : e_2$: the branch e_1 is selected only if the function is applied to a value of type `Int`, that is, if the type α of x is instantiated to `Int` (notice that when typing the body of the function Δ contains only α). More generally, the branch e_1 in $e{\in}t\,?\,e_1 : e_2$ can be selected only if e can return a value in t, that is to say, if there exists a substitution σ for any type variables *even those in* Δ such as the intersection of $t'\sigma$ and t is not empty (t is a closed, so $t\sigma = t$). Therefore, in order to achieve maximum precision the rule (*case*) must check $\Delta\,;\Gamma \vdash e_1 : s$ only if there exists σ such that $t'\sigma\wedge t \not\simeq \mathbb{0}$. Since $t'\leq\neg t$ (strictly) implies that for all substitutions σ, $t'\sigma{\leq}\neg t$ (recall that t is a closed type), then by the contrapositive the existence of a substitution σ such that $t'\sigma\not\leq\neg t$ implies $t'\not\leq\neg t$. The latter is equivalent to $t'\wedge t\not\simeq\mathbb{0}$: the intersection of t and t' is not empty. So we slightly over-approximate the test of selection and check the type of e_1 under the weaker hypothesis $t'\wedge t\not\simeq\mathbb{0}$ which ensures that the typing will hold also under the stronger (and sought) hypothesis that there exists σ such that $t'\sigma\wedge t\not\simeq\mathbb{0}$ (the difference only matters with some specific cases involving *indivisible* types: see [6]).

Notice that explicit type-substitutions are only needed to type applications of polymorphic functions. Since no such application occurs in the bodies of `map` and `even` as defined in Section 2 (the m and f inside the body of `map` are abstracted variables and, thus, have monomorphic types), then they can be typed by this system as they are (as long as they are not applied one to the other there is no need to infer any set of type-substitutions). So we can already see that our language passes the second test, namely, that `map` and `even` have the types declared in their signatures. Let us detail just the most interesting case, that is, the typing of the term `even` defined in equation (11) (even though the typing of the type-case in (12), the term defining `map`, is interesting, as well). According to

the rule (*abstr*) we have to check that under the hypothesis x:Int the expression $x\in$Int?$(x \bmod 2) = 0 : x$ has type Bool, and that under the hypothesis $x : \alpha\backslash$Int the same expression has type $\alpha\backslash$Int. So we have two distinct applications of the (*case*) rule. In one x is of type Int, thus the check Int $\not\leq$ Int fails, and therefore only the first branch, $(x \bmod 2) = 0$, is type checked (the second is skipped). Since under the hypothesis x : Int the expression $(x \bmod 2) = 0$ has type Bool, then so has the whole type-case expression. In the other application of (*case*), x is of type $\alpha\backslash$Int, so the test $\alpha\backslash$Int $\not\leq \neg$Int clearly fails, and only the second branch is checked (the first is skipped). Since this second branch is x, then the whole type-case expression has type $\alpha\backslash$Int, as expected. This example shows two important aspects of our typing rules. First, it shows the importance of Δ to record monomorphic variables, since it may contain some variables that do not occur in Γ. For instance, when typing the first branch of even, the type environment contains only x : Int but Δ is $\{\alpha\}$ and this forbids to consider α as polymorphic (if we allowed to instantiate any variable that does not occur in Γ, then the term obtained from the even function (11) by replacing the first branch by $(\lambda^{\alpha\to\alpha} y.y)[\{\text{Bool}/\alpha\}]$true would be well-typed, which is wrong since α is monomorphic in the body of even). Second, this example shows why if in some application of the (*case*) rule a branch is not checked, then the type checking of the whole type-case expression must not necessarily fail: the well-typing of this branch may be checked under different hypothesis (typically when occurring in the body of an overloaded function).[4] The reader can refer to Section 3.3 of [12] for a more detailed discussion on this point.

We conclude the presentation of the type system with two observations.

First, we said at the beginning that we consider only judgments $\Delta \fatsemi \Gamma \vdash e : t$ where the type variables occurring in Γ are contained in Δ. The intuition is that the types in Γ are the types of the formal parameters of some outer functions and therefore any type variable occurring in them must be considered monomorphic —*ie*, must be contained in Δ. Without such a requirement it would be possible to have a derivation such as the following one:

$$\frac{\overline{\{\alpha\} \fatsemi (x : \beta) \vdash x : \beta} \qquad \{\gamma/\beta\} \sharp \{\alpha\}}{\{\alpha\} \fatsemi (x : \beta) \vdash x[\{\gamma/\beta\}] : \gamma}$$

which states that by assuming that x has (any) type β, we can infer that it has also (any other) type γ, which is unsound. The condition $\mathsf{var}(\Gamma) \subseteq \Delta$ is preserved by the typing rules (see Lemma B.1) and we always start with Γ and Δ satisfying the condition (typically, when we type a closed expression we start by $\Delta = \Gamma = \emptyset$), therefore, henceforth, we implicitly assume the condition $\mathsf{var}(\Gamma) \subseteq \Delta$ to hold in all judgments we consider.

Second, the rule (*subsum*) makes the type system dependent on the subtyping relation $\leq$ defined in [6]. It is important not to confuse the subtyping relation $\leq$ of our system, which denotes semantic subtyping (*ie*, set-theoretic inclusion of denotations), with the one typically used in the type reconstruction systems for ML, which stands for type variable instantiation. For example, in ML we have $\alpha \to \alpha \leq$ Int $\to$ Int (because Int $\to$ Int is an instance of $\alpha \to \alpha$). But this is not true in our system, as the relation must hold for *every possible instantiation of* α, thus in particular for α equal to Bool. In the companion paper [5] we define the preorder $\sqsubseteq_\Delta$ which includes the type variable instantiation of the preorder typically used for ML, so any direct comparison with constraint systems for ML types should focus on $\sqsubseteq_\Delta$ rather than $\leq$ and it can be found in the companion paper [5].

3.4 Type soundness

Subject reduction and progress properties hold for this system.

Theorem 3.3 (Subject Reduction)**.** *For every term e and type t, if $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.*

Theorem 3.4 (Progress)**.** *Let e be a well-typed closed term. If e is not a value, then there exists a term e' such that $e \rightsquigarrow e'$.*

The proofs of both theorems, though unsurprising, are rather long and technical and can be found in Appendix B.2. They allow us to conclude that the type system is sound.

Corollary 3.5 (Type soundness)**.** *Let e be a well-typed closed expression, that is, $\vdash e : t$ for some t. Then either e diverges or it returns a value of type t.*

3.5 Expressing intersection type systems

We can now state the first stand-alone theoretical contribution of our work. Consider the sub-calculus of our calculus in which type-substitutions occur only in decorations and without constants and type-case expressions, that is,

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i\to t_i}_{[\sigma_j]_{j\in J}} x.e \tag{20}$$

and whose types are *inductively* produced by the grammar

$$t ::= \alpha \mid t \to t \mid t \wedge t$$

This calculus is closed with respect to β-reduction as defined by the reduction rule (18) (without type-cases, union and negation types are not needed). It constitutes an explicitly-typed λ-calculus with intersection types whose expressive power subsumes that of classic intersection type systems (without an universal element ω, of course), as expressed by the following theorem.

Theorem 3.6. *Let $\vdash_{BCD}$ denote provability in the Barendregt, Coppo, and Dezani system [1], and $\lceil e \rceil$ be the pure λ-calculus term obtained from e by erasing all types occurring in it.*

If $\vdash_{BCD} m : t$, then $\exists e$ such that $\vdash e : t$ and $\lceil e \rceil = m$.

Therefore, this sub-calculus solves a longstanding open problem, that is the definition of explicit type annotations for λ-terms in intersection type systems, without any further syntactic modification. See Section 7 on related work for an extensive comparison.

The proof of Theorem 3.6 is constructive (*cf.*, Appendix B.3). Therefore we can transpose decidability results of intersection type systems to our system. In particular, type reconstruction[5] for the subcalculus (20) is undecidable and this implies the undecidability of type reconstruction for the whole calculus *without recursive types* (with recursive types type reconstruction is trivially decidable since every λ-term can be typed by the recursive type $\mu X.(X\to X)\vee *$). In Section 4 we prove that type inference for our system is decidable. The problem of reconstructing type-substitutions (*ie*, given a term of grammar (3), deciding whether it is possible to add sets of type-substitutions in it so that it becomes a well-typed term of our calculus) is dealt with in the companion paper [5].

3.6 Elimination of sets of type-substitutions

To compare with existing intersection type systems, the calculus in (20) includes neither type-cases nor expressions of the form

[4] From a programming language point of view it is important to check that during type checking every branch of a given type-case expression is checked —*ie*, it can be selected— at least once. This corresponds to checking the absence of redundant cases in pattern matching. We omitted this check since it is not necessary for formal development.

[5] We recall that *type reconstruction* is the problem of finding whether there exists a type-annotation that makes a given expression well-typed; *type inference* is the problem of checking whether an expression is well-typed, and *type checking* is the problem of checking whether an expression has a given type.

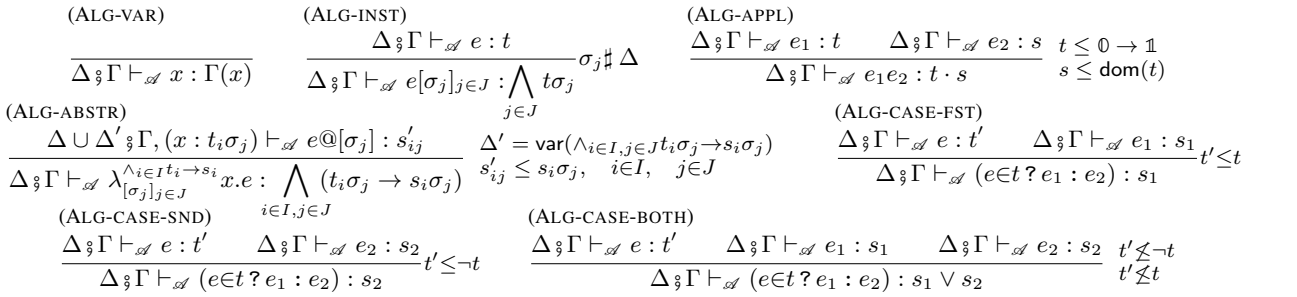

Figure 2. Typing algorithm

$e[\sigma_j]_{j\in J}$. While it is clear that type-cases increase the expressive power of the calculus, one may wonder whether the same is true for $e[\sigma_j]_{j\in J}$. In this section, we prove that the terms of the form $e[\sigma_j]_{j\in J}$ are redundant insofar as their presence in the calculus does not increase its expressive power. Consider the subcalculus whose terms are

$$e ::= x \mid ee \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e \in t\,?\,e : e \qquad (21)$$

that is, the calculus in which sets of type-substitutions appear only in decorations. Consider the embedding $[\![.]\!]$ of our calculus (16) into this subcalculus, defined as

$$[\![e[\sigma_j]_{j\in J}]\!] = e@[\sigma_j]_{j\in J}$$

as the identity for variables, and as its homomorphic propagation for all the other expressions. Then it is easy to prove the following theorem

Theorem 3.7. *For every well-typed expression e:*
1. $e \rightsquigarrow^* v \Rightarrow [\![e]\!] \rightsquigarrow^* [\![v]\!]$,
2. $[\![e]\!] \rightsquigarrow^* v \Rightarrow e \rightsquigarrow^* v'$ *and* $v = [\![v']\!]$

meaning that the subcalculus defined above is equivalent to the full calculus. Although expressions of the form $e[\sigma_j]_{j\in J}$ do not bring any further expressive power, they play a crucial role in local type inference, which is why we included them in our calculus. As we explain in details in the companion paper, for local type inference we need to reconstruct sets of type-substitutions that are applied to expressions but we *must not* reconstruct sets of type-substitutions that are decorations of λ-expressions. The reason is pragmatic and can be shown by considering the following two terms: $(\lambda^{\alpha\to\alpha} x.x)3$ and $(\lambda^{\alpha\to\alpha} x.4)3$. Every functional programmer will agree that the first expression must be considered well-typed while the second must not, for the simple reason that the constant function $(\lambda^{\alpha\to\alpha} x.4)$ does not have type $\alpha \to \alpha$. Indeed in the first case it is possible to apply a set of type-substitutions that makes the term well typed, namely $(\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/\alpha\}]3$, while no such application exists for the second term. However, if we allowed reconstruction also for decorations, then the second term could be made well typed by adding the following decoration $(\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} x.4)3$. In conclusion, for type inference it is important to keep the expression $e[\sigma_j]_{j\in J}$, since well-typing of $e@[\sigma_j]_{j\in J}$ does not imply that of $e[\sigma_j]_{j\in J}$.

4. Typing algorithm

The rules in Figure 1 do not describe a typing algorithm since they are not syntax directed. As customary the problem is the subsumption rule, and the way to go is to eliminate this rule by embedding appropriate checks of the subtyping relation into the rules that need it. This results in the system formed by the rules of Figure 2. This system is algorithmic (as stressed by $\vdash_{\mathscr{A}}$): in every case at most one rule applies, either because of the syntax of the term or because of mutually exclusive side conditions. Subsumption is no longer present and, instead, subtype checking has been pushed in all the remaining rules.

The rule for type-cases has been split in three rules (plus a fourth uninteresting rule we omitted that states that when $e : \mathbb{0}$ —*ie*, it is the provably diverging expression— then the whole type-case expression has type $\mathbb{0}$) according to whether one or both branches can be selected. Here the only modification is in the case where both branches can be selected: in the rule (*case*) in Figure 1 the types of the two branches were subsumed to a common type s, while (ALG-CASE-BOTH) returns the least upper bound (*ie*, the union) of the two types.

The rule for abstractions underwent a minor modification with respect to the types returned for the body, which before were subsumed to the type declared in the interface while now the subtyping relation $s'_{ij} \leq s_i\sigma_j$ is explicitly checked.

The elimination of the subsumption yields a simplification in typing the application of type-substitutions, since in the system of Figure 1 without subsumption every premise of an (*inter*) rule is the consequence of an (*inst*) rule. The two rules can thus be merged into a single one, yielding the (ALG-INST) rule (see Appendix C.1 and in particular Theorem C.1).

As expected, the core of the typing algorithm is the rule for application. In the system of Figure 1, in order to apply the (*appl*) rule, the type of the function had to be subsumed to an arrow type, and the type of the argument had to be subsumed to the domain of that arrow type; then the co-domain of the arrow is taken to type the application. In the algorithmic rule (ALG-APPL), this is done by the type meta-operator "$\cdot$" which is formally defined as follows: $t \cdot s \stackrel{\text{def}}{=} \min\{u \mid t \leq s \to u\}$. In words, if t is the type of the function and s the type of the argument, this operator looks for the smallest arrow type larger than t and with domain s, and it returns its co-domain. More precisely, when typing $e_1 e_2$, the rule (ALG-APPL) checks that the type t of e_1 is a functional one (*ie*, $t \leq \mathbb{0} \to \mathbb{1}$). It also checks that the type s of e_2 is a subtype of the domain of t (denoted by $\mathsf{dom}(t)$). Because t is not necessarily an arrow type (in general, it is equivalent to a disjunctive normal form like the one of equation (15) in Section 3), the definition of the domain is not immediate. The domain of a function whose type is an intersection of arrows and negation of arrows is the union of the domains of all positive literals. For instance the domain of a function of type $(\mathtt{Int}\to\mathtt{Int}) \wedge (\mathtt{Bool}\to\mathtt{Bool})$ is $\mathtt{Int} \vee \mathtt{Bool}$, since it can be equally applied to integer or Boolean arguments, while the domain of `even` as defined in (11) is $\mathtt{Int} \vee (\alpha \setminus \mathtt{Int})$, that is $\mathtt{Int} \vee \alpha$. The domain of a union of functional types is the intersection of each domain. For instance an expression of type $(s_1\to s_2) \vee (t_1\to t_2)$ will return either a function of type $s_1\to s_2$ or a function of type $t_1\to t_2$, so this expression can be applied only to arguments that fit both cases, that is, to arguments in $s_1 \wedge t_1$. Formally, if $t \leq \mathbb{0}\to\mathbb{1}$, then $t \simeq \bigvee_{i\in I}(\bigwedge_{p\in P_i}(s_p\to t_p) \wedge \bigwedge_{n\in N_i} \neg(s_n\to t_n) \wedge \bigwedge_{q\in Q_i} \alpha_q \wedge \bigwedge_{r\in R_i} \neg\beta_r)$ (with all the P_i's not empty), and therefore $\mathsf{dom}(t) \stackrel{\text{def}}{=} \bigwedge_{i\in I} \bigvee_{p\in P_i} s_p$ (here type

variables do not count since they are intersected and universally quantified so the definition of the domain must hold also when their intersection is $\mathbb{1}$). Finally, the type returned in (ALG-APP) is $t \cdot s$, which we recall is the smallest result type that can be obtained by subsuming t to an arrow type compatible with s. We can prove that for every type t such that $t \leq \mathbb{0}\to\mathbb{1}$ and type s such that $s \leq \mathsf{dom}(t)$, the type $t \cdot s$ exists and can be effectively computed (see Lemma C.12).

The algorithmic system is sound and complete with respect to the type system of Figure 1 and satisfies the minimum typing property (see Appendix C for the proofs).

Theorem 4.1 (Soundness). *If $\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t$, then $\Delta \wr \Gamma \vdash e : t$.*

Theorem 4.2 (Completeness). *If $\Delta \wr \Gamma \vdash e : t$, then there exists a type s such that $\Delta \wr \Gamma \vdash_{\mathscr{A}} e : s$ and $s \leq t$.*

Corollary 4.3 (Minimum typing). *If $\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t$, then $t = \min\{s \mid \Delta \wr \Gamma \vdash e : s\}$.*

Finally, it is quite easy to prove that type inference is decidable. It suffices to define the *size* of an expression as follows:

$$\begin{aligned}
\mathrm{size}(x) &= 1\\
\mathrm{size}(e_1e_2) &= \mathrm{size}(e_1) + \mathrm{size}(e_2) + 1\\
\mathrm{size}(\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e) &= \mathrm{size}(e) + 1\\
\mathrm{size}(e\in t\,?\,e_1 : e_2) &= \mathrm{size}(e) + \mathrm{size}(e_1) + \mathrm{size}(e_2) + 1\\
\mathrm{size}(e[\sigma_j]_{j\in J}) &= \mathrm{size}(e) + 1
\end{aligned}$$

and show that the expressions occurring in the premises of every rule of the algorithm are strictly smaller than the expression in its conclusion. We can then deduce the termination of the type inference algorithm.

Theorem 4.4 (Termination). *Let e be an expression. Then the type inference algorithm for e terminates.*

This system constitutes a further theoretical contribution of our work since with this type system the language defined by grammar (16), the one by grammar (21), and *a fortiori* the one by grammar (20) are intersection type systems that all satisfy the Curry-Howard isomorphism since there is a one-to-one correspondence between terms and proofs of the algorithmic system.

5. Evaluation

In this section we define an efficient execution model for the polymorphic calculus as a conservative extension of the execution model of the monomorphic calculus: by "efficient" we mean that monomorphic expressions will be evaluated as efficiently as in the original $\mathbb{C}$Duce runtime. In fact, even polymorphic expressions will be evaluated as efficiently as well (as if type variables were basic monomorphic types) despite the fact that the formal reduction semantics of polymorphic expressions includes a run-time relabeling operation. The key observation that allows us to define an efficient execution model for the polymorphic calculus is that relabeling can be implemented *lazily* so that the only case in which relabeling is computed at run-time will correspond to testing the type of a partial application of a polymorphic function. In practice, this case is so rare —at least in the XML setting— that there is no difference between monomorphic and polymorphic evaluation.

5.1 Monomorphic Language

Let us start by recalling the execution model of monomorphic $\mathbb{C}$Duce, which is a classic closure-based evaluation. Expressions and values are defined as

$$\begin{aligned}
e &::= c \mid x \mid \lambda^t x.e \mid ee \mid e \in s\,?\,e : e\\
v &::= c \mid \langle \lambda^t x.e, \mathscr{E}\rangle
\end{aligned}$$

where t denotes an intersection of arrow types, s denotes a closed type, and $\mathscr{E}$ denotes an *environment*, that is, a substitution mapping expression variables into values. The big step semantics is:

$$\text{(ME-Const)}\quad \mathscr{E} \vdash_{\mathsf{m}} c \Downarrow c \qquad \text{(ME-Var)}\quad \mathscr{E} \vdash_{\mathsf{m}} x \Downarrow \mathscr{E}(x) \qquad \text{(ME-Closure)}\quad \mathscr{E} \vdash_{\mathsf{m}} \lambda^t x.e \Downarrow \langle \lambda^t x.e, \mathscr{E}\rangle$$

$$\text{(ME-Apply)}\quad \frac{\mathscr{E} \vdash_{\mathsf{m}} e_1 \Downarrow \langle \lambda^t x.e, \mathscr{E}'\rangle \qquad \mathscr{E} \vdash_{\mathsf{m}} e_2 \Downarrow v_0 \qquad \mathscr{E}', x \mapsto v_0 \vdash_{\mathsf{m}} e \Downarrow v}{\mathscr{E} \vdash_{\mathsf{m}} e_1e_2 \Downarrow v}$$

$$\text{(ME-Type Case T)}\quad \frac{\mathscr{E} \vdash_{\mathsf{m}} e_1 \Downarrow v_0 \qquad v_0 \in_{\mathsf{m}} t \qquad \mathscr{E} \vdash_{\mathsf{m}} e_2 \Downarrow v}{\mathscr{E} \vdash_{\mathsf{m}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$\text{(ME-Type Case F)}\quad \frac{\mathscr{E} \vdash_{\mathsf{m}} e_1 \Downarrow v_0 \qquad v_0 \notin_{\mathsf{m}} t \qquad \mathscr{E} \vdash_{\mathsf{m}} e_3 \Downarrow v}{\mathscr{E} \vdash_{\mathsf{m}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

To complete the definition we define the relation $v \in_{\mathsf{m}} t$, that is, membership of a (monomorphic) value to a (monomorphic) type:

$$\begin{aligned}
c \in_{\mathsf{m}} t &\stackrel{\text{def}}{\iff} b_c \leq t\\
\langle \lambda^s x.e, \mathscr{E}\rangle \in_{\mathsf{m}} t &\stackrel{\text{def}}{\iff} s \leq t
\end{aligned}$$

where $\leq$ is the subtyping relation of $\mathbb{C}$Duce [12].

5.2 Polymorphic Language

In the naive extension of this semantics to the explicitly-typed polymorphic calculus of Section 3, we deal with type-substitutions as we do for environments, that is, by storing them in closures. This is reflected by the following definition where, for brevity, we write σ_I to denote the set of type-substitutions $[\sigma_i]_{i\in I}$:

$$\begin{aligned}
e &::= c \mid x \mid \lambda^t_{\sigma_I} x.e \mid ee \mid e \in t\,?\,e : e \mid e\sigma_I\\
v &::= c \mid \langle \lambda^t_{\sigma_I} x.e, \mathscr{E}, \sigma_I\rangle
\end{aligned}$$

The big-step semantics is then defined as follows, where each expression is evaluated with respect to an environment $\mathscr{E}$ determining the current value substitutions and a set of type-substitutions σ_I:

$$\text{(PE-Const)}\quad \sigma_I; \mathscr{E} \vdash_{\mathsf{p}} c \Downarrow c \qquad \text{(PE-Var)}\quad \sigma_I; \mathscr{E} \vdash_{\mathsf{p}} x \Downarrow \mathscr{E}(x)$$

$$\text{(PE-Closure)}\quad \sigma_I; \mathscr{E} \vdash_{\mathsf{p}} \lambda^t_{\sigma_J} x.e \Downarrow \langle \lambda^t_{\sigma_J} x.e, \mathscr{E}, \sigma_I\rangle \qquad \text{(PE-Instance)}\quad \frac{\sigma_I \circ \sigma_J; \mathscr{E} \vdash_{\mathsf{p}} e \Downarrow v}{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e\sigma_J \Downarrow v}$$

$$\text{(PE-Apply)}\quad \frac{\begin{array}{c}\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_1 \Downarrow \langle \lambda^{\wedge_{l\in L} s_l\to t_l}_{\sigma_K} x.e, \mathscr{E}', \sigma_H\rangle \qquad \sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_2 \Downarrow v_0\\ \sigma_J = \sigma_H \circ \sigma_K \quad P = \{j \in J \mid \exists l \in L : v_0 \in_{\mathsf{p}} s_l\sigma_j\}\\ \sigma_P; \mathscr{E}', x \mapsto v_0 \vdash_{\mathsf{p}} e \Downarrow v\end{array}}{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_1e_2 \Downarrow v}$$

$$\text{(PE-Type Case T)}\quad \frac{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_1 \Downarrow v_0 \qquad v_0 \in_{\mathsf{p}} t \qquad \sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_2 \Downarrow v}{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

$$\text{(PE-Type Case F)}\quad \frac{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_1 \Downarrow v_0 \qquad v_0 \notin_{\mathsf{p}} t \qquad \sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_3 \Downarrow v}{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

The membership relation $v \in_{\mathsf{p}} t$ for polymorphic values is inductively defined as:

$$\begin{aligned}
c \in_{\mathsf{p}} t &\stackrel{\text{def}}{\iff} b_c \leq t\\
\langle \lambda^s_{\sigma_J} x.e, \mathscr{E}, \sigma_I\rangle \in_{\mathsf{p}} t &\stackrel{\text{def}}{\iff} s(\sigma_I \circ \sigma_J) \leq t
\end{aligned}$$

where $\leq$ is the subtyping relation of Castagna and Xu [6]. It is not difficult to show that this big-step semantics is equivalent to the small-step one of Section 3. Let $(\!|.|\!)$ be the transformation that maps values of the polymorphic language into corresponding values of the calculus, that is

$$(\!|c|\!) = c \qquad \text{and} \qquad (\!|\langle \lambda^s_{\sigma_J} x.e, \mathscr{E}, \sigma_I\rangle|\!) = \lambda^s_{\sigma_I\circ\sigma_J} x.(e(\!|\mathscr{E}|\!)) \quad (22)$$

where $(\!|\mathscr{E}|\!)$ applies $(\!|.|\!)$ to all the values in the range of $\mathscr{E}$. Let $\mathbb{i}$ denote the singleton set containing the empty type-substitution $[\{\,\}]$, which is the neutral element of the composition of sets of type-substitutions. Then we have:

Theorem 5.1. *Let e be a well-typed closed explicitly-typed expression ($\vdash_{\mathscr{A}} e : t$). Then:*

$$\mathbb{i};\varnothing \vdash_{\mathsf{p}} e \Downarrow v \iff e \rightsquigarrow^* (\!|v|\!)$$

This implementation has a significant computational burden compared to that of the monomorphic language: first of all, each application of (PE-APPLY) computes the set P, which requires to implement several type-substitutions and membership tests; second, each application of (PE-INSTANCE) computes the composition of two sets of type-substitutions. In the next section we describe a different solution consisting in the compilation of the explicitly typed calculus into an intermediate language so that these computations are postponed as much as possible and are performed only if and when they are really necessary.

5.3 Intermediate Language

The intermediate language into which we compile the explicitly-typed polymorphic language is very similar to the monomorphic version. The only difference is that λ-abstractions (both in expressions and closures) may contain type variables in their interface and have an extra decoration Σ which is a term *denoting* a set of type-substitutions.

$$\begin{array}{rcl} e & ::= & c \mid x \mid \lambda^t_\Sigma x.e \mid ee \mid e \in t\,?\,e : e \\ v & ::= & c \mid \langle \lambda^t_\Sigma x.e, \mathscr{E}\rangle \\ \Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma,\Sigma') \mid \mathtt{sel}(x,t,\Sigma) \end{array}$$

Intuitively, a $\mathtt{comp}(\Sigma,\Sigma')$ term corresponds to an application of the $\circ$ composition operator to the sets of type substitutions denoted by Σ and Σ', while a $\mathtt{sel}(x,t,\Sigma)$ term selects the subset of type substitutions σ denoted by Σ that are compatible with the fact that (the value instantiating) x belongs to the domain of $t\sigma$.

The big step semantics for this intermediate language is:

(OE-CONST) $\mathscr{E} \vdash_{\mathsf{o}} c \Downarrow c$

(OE-VAR) $\mathscr{E} \vdash_{\mathsf{o}} x \Downarrow \mathscr{E}(x)$

(OE-CLOSURE) $\mathscr{E} \vdash_{\mathsf{o}} \lambda^t_\Sigma x.e \Downarrow \langle \lambda^t_\Sigma x.e, \mathscr{E}\rangle$

(OE-APPLY)
$$\frac{\mathscr{E} \vdash_{\mathsf{o}} e_1 \Downarrow \langle \lambda^t_\Sigma x.e, \mathscr{E}'\rangle \quad \mathscr{E} \vdash_{\mathsf{o}} e_2 \Downarrow v_0 \quad \mathscr{E}', x \mapsto v_0 \vdash_{\mathsf{o}} e \Downarrow v}{\mathscr{E} \vdash_{\mathsf{o}} e_1 e_2 \Downarrow v}$$

(OE-TYPE CASE T)
$$\frac{\mathscr{E} \vdash_{\mathsf{o}} e_1 \Downarrow v_0 \quad v_0 \in_{\mathsf{o}} t \quad \mathscr{E} \vdash_{\mathsf{o}} e_2 \Downarrow v}{\mathscr{E} \vdash_{\mathsf{o}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

(OE-TYPE CASE F)
$$\frac{\mathscr{E} \vdash_{\mathsf{o}} e_1 \Downarrow v_0 \quad v_0 \notin_{\mathsf{o}} t \quad \mathscr{E} \vdash_{\mathsf{o}} e_3 \Downarrow v}{\mathscr{E} \vdash_{\mathsf{o}} e_1 \in t\,?\,e_2 : e_3 \Downarrow v}$$

Notice that this semantics is structurally the same as that of the monomorphic language. There are only two minor differences: (i) λ-abstractions have an extra decoration Σ (which has no impact on efficiency since it corresponds in the implementation to manipulate descriptors with an extra field) and (ii) the corresponding (_E-TYPE CASE) rules use a slightly different relation: $\in_{\mathsf{o}}$ instead of $\in_{\mathsf{m}}$. It is thus easy to see that in terms of steps of reduction the two semantics have the same complexity. More precisely, if you take a term of the monomorphic calculus and a term of the intermediate language with the same erasure and that select the same branches of the typecases, then they perform exactly the same reduction. What changes is the test of the membership relation ($\in_{\mathsf{o}}$ rather than $\in_{\mathsf{m}}$) since, when the value to be tested is a closure, we need to materialize relabelings. In other words, we have to evaluate the Σ expression decorating the function and apply the resulting set of substitutions to the interface of the function. Formally:

$$\begin{array}{rcl} c \in_{\mathsf{o}} t & \stackrel{\text{def}}{\iff} & b_c \le t \\ \langle \lambda^{\mathsf{s}}_\Sigma x.e, \mathscr{E}\rangle \in_{\mathsf{o}} t & \stackrel{\text{def}}{\iff} & s(\mathsf{eval}(\mathscr{E},\Sigma)) \le t \end{array}$$

where the evaluation of the symbolic set of type-substitutions is inductively defined as

$$\begin{array}{rcl} \mathsf{eval}(\mathscr{E},\sigma_I) &=& \sigma_I \\ \mathsf{eval}(\mathscr{E},\mathtt{comp}(\Sigma,\Sigma')) &=& \mathsf{eval}(\mathscr{E},\Sigma)\circ\mathsf{eval}(\mathscr{E},\Sigma') \\ \mathsf{eval}(\mathscr{E},\mathtt{sel}(x,\bigwedge_{i\in I} t_i{\to}s_i,\Sigma)) &=& \\ & & [\ \sigma_j \in \mathsf{eval}(\mathscr{E},\Sigma) \mid \exists i{\in}I : \mathscr{E}(x) \in_{\mathsf{o}} t_i\sigma_j\ ] \end{array}$$

Notice in the last rule the crucial role played by x and $\mathscr{E}$: by using an expression variable x in the symbolic representation of type-substitutions and relying on its interpretation through $\mathscr{E}$, we have transposed to type-substitutions the same benefits that closures bring to value substitutions: just as closures allow *value*-substitutions to be materialized only when a formal parameter is used rather than at the moment of the reduction, so our technique allows *type*-substitutions to be materialized only when a type variable is effectively tested, rather than at the moment of the reduction.

It is easy to see that the only case in which the computation of $\in_{\mathsf{o}}$ is more expensive than that of $\in_{\mathsf{m}}$ is when the value whose type is tested is a closure $\langle \lambda^t_\Sigma x.e, \mathscr{E}\rangle$ in which t is not closed and Σ is not $\mathbb{i}$.[6] The Σ decoration is different from $\mathbb{i}$ only if the closure is the result of a partial application of a curried function. The type t is not closed only if such partial application yielded a polymorphic function. In conclusion, the evaluation of an expression in the polymorphic language is more expensive than the evaluation of a similar[7] expression of the monomorphic language only if it tests the type of a polymorphic function resulting from the partial application of a polymorphic curried function. The additional overhead is limited only to this particular test and in all the other cases the evaluation is as efficient as in the monomorphic case. Finally, it is important to stress that this holds true also if we add product types: the test of a pair of values in the polymorphic case is as expensive as in the monomorphic case and so is the rest of the evaluation. Since in the XML setting the vast majority of the computation time is spent in testing products (since they encode sequences, trees, and XML elements), then the overhead brought by adding polymorphism —*ie*, the overhead due to testing the type of a polymorphic partial application of a polymorphic curried function— is negligible in practice.

All that remains to do is to define the compilation of the explicitly-typed language into the intermediate language:

$$\begin{array}{rcl} [\![x]\!]_\Sigma &=& x \\ [\![\lambda^t_{\sigma_I} x.e]\!]_\Sigma &=& \lambda^t_{\mathtt{comp}(\Sigma,\sigma_I)} x.[\![e]\!]_{\mathtt{sel}(x,t,\mathtt{comp}(\Sigma,\sigma_I))} \\ [\![e_1 e_2]\!]_\Sigma &=& [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\ [\![e\sigma_I]\!]_\Sigma &=& [\![e]\!]_{\mathtt{comp}(\Sigma,\sigma_I)} \\ [\![e_1 \in t\,?\,e_2 : e_3]\!]_\Sigma &=& [\![e_1]\!]_\Sigma \in t\,?\,[\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma \end{array}$$

Given a closed program e we compile it in the intermediate language as $[\![e]\!]_{\mathbb{i}}$. In practice, the compilation will be even simpler since we apply it only to expressions generated by local type inference algorithm described in the companion paper where all λ's are decorated by $\mathbb{i}$ (*cf.* discussion at the end of Section 3.6). So the second case of the definition simplifies to:

$$[\![\lambda^t_{\mathbb{i}} x.e]\!]_\Sigma \quad = \quad \lambda^t_\Sigma x.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma)}$$

The compilation is adequate:

Theorem 5.2. *Let e be a well-typed closed explicitly-typed expression ($\vdash_{\mathscr{A}} e : t$). Then*

$$\mathbb{i};\varnothing \vdash_{\mathsf{p}} e \Downarrow v \iff \vdash_{\mathsf{o}} [\![e]\!]_{\mathbb{i}} \Downarrow v'$$

with $(\!|v|\!) = (\!|v'|\!)$.

where $(\!|\langle \lambda^t_\Sigma x.e, \mathscr{E}\rangle|\!)$ evaluates all the symbolic expressions and type-substitutions in the term (see (22)). By combining Theorems 5.1 and 5.2 we obtain the adequacy of the compilation:

[6] To be more precise, when there exists a substitution $\sigma \in \mathsf{eval}(\mathscr{E},\Sigma)$ such that $\mathsf{var}(t) \cap \mathsf{dom}(\sigma) \neq \varnothing$. Notice that the tests of the subtyping relation for monomorphic and polymorphic types have the same complexity [6].

[7] By similar we intend with the same syntax tree but only closed types.

Corollary 5.3. *Let e be a well-typed closed explicitly-typed expression ($\vdash_{\mathscr{A}} e : t$). Then*

$$\vdash_{\mathsf{o}} [\![e]\!]_{\mathsf{i}} \Downarrow v \iff e \rightsquigarrow^* (\!|v|\!)$$

Finally, let us come back to the membership relation for function types, namely:

$$\langle \lambda^{\mathsf{s}}_{\Sigma} x.e, \mathscr{E} \rangle \in_{\mathsf{o}} t \quad \stackrel{\text{def}}{\iff} \quad s(\mathsf{eval}(\mathscr{E}, \Sigma)) \leq t$$

In Footnote 6 we signalled that the only case in which this test is more expensive than in the monomorphic case is when we have to evaluate $\mathsf{eval}(\mathscr{E}, \Sigma)$ and that this may be necessary only if there exists $\sigma \in \mathsf{eval}(\mathscr{E}, \Sigma)$ such that $\mathsf{var}(s) \cap \mathsf{dom}(\sigma) \neq \varnothing$ (this includes the case of functions that are result of a partial application of a polymorphic function). Of course we cannot perform this test without evaluating the eval expression. We can however overapproximate the domains of the various σ produced by $\mathsf{eval}(\mathscr{E}, \Sigma)$ by using the domain of Σ (the difference being that we consider also the substitutions σ that would be discarded by a selection). Notice that all the expressions Σ denoting sets of substitutions are statically generated by our compilation and are not modified at run-time. This means that for every function value we can statically decide whether the condition $\mathsf{var}(s) \cap \mathsf{dom}(\Sigma) \neq \varnothing$ is satisfied or not. This test soundly approximates for every $\mathscr{E}$ the condition whether there exists $\sigma \in \mathsf{eval}(\mathscr{E}, \Sigma)$ such that $\mathsf{var}(s) \cap \mathsf{dom}(\sigma) \neq \varnothing$ where $\mathsf{dom}(\Sigma)$ is defined as follows:

$$\begin{array}{rcl} \mathsf{dom}(\sigma_I) & = & \bigcup_{i \in I} \mathsf{dom}(\sigma_i) \\ \mathsf{dom}(\mathtt{comp}(\Sigma, \Sigma')) & = & \mathsf{dom}(\Sigma) \cup \mathsf{dom}(\Sigma') \\ \mathtt{sel}(x, t, \Sigma)) & = & \mathsf{dom}(\Sigma) \end{array}$$

In practice, we can modify our compilation technique to flag (*eg*, by "λ") the functions which may require the evaluation of $\mathsf{eval}(,)$, as follows:

$$[\![\lambda^t_{\mathsf{i}} x.e]\!]_{\Sigma} \quad = \quad \begin{cases} \lambda^t_{\Sigma} x.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma)} & \text{if } \mathsf{var}(t) \cap \mathsf{dom}(\Sigma) = \varnothing \\ \lambda\!\!\!^{\,t}_{\Sigma} x.[\![e]\!]_{\mathtt{sel}(x,t,\Sigma)} & \text{otherwise} \end{cases}$$

and then evaluate the symbolic substitutions only for marked functions:

$$\begin{array}{lcl} \langle \lambda^{\mathsf{s}}_{\Sigma} x.e, \mathscr{E} \rangle \in_{\mathsf{o}} t & \stackrel{\text{def}}{\iff} & s \leq t \\ \langle \lambda\!\!\!^{\,\mathsf{s}}_{\Sigma} x.e, \mathscr{E} \rangle \in_{\mathsf{o}} t & \stackrel{\text{def}}{\iff} & s(\mathsf{eval}(\mathscr{E}, \Sigma)) \leq t \end{array}$$

5.4 Let-polymorphism

A function is polymorphic if it can be safely applied to arguments of different types. The calculus presented supports a varied palette of different forms of polymorphism: it uses subtype polymorphism (a function can be applied to arguments whose types are subtypes of its domain type), the combination of intersection types and typecase expressions yields "ad hoc" polymorphism (aka overloading), and finally the use of type variables in function interfaces provides parametric polymorphism. Polymorphism is interesting when used with bindings: instead of repeating the definition of a function every time we need to apply it, it is more convenient to define the function once, bind it to an expression variable, and use the variable every time we need to apply the function. In the current system it is possible to combine binding only with the first two kinds of polymorphism: different occurrences of a variable bound to a function can be given different types —thus, be applied to arguments of different types—, either by subsumption (*ie*, by assigning to the variable a super-type of the type of the function it denotes) or by intersection elimination (*ie*, by assigning to a variable one of the types that form the intersection type of the function it denotes). However, as it is well known in the languages of the ML-family, in the current setting it is not possible to combine binding and parametric polymorphism. Distinct occurrences of a variable cannot be given different types by instantiation (*ie*, by assigning to the variable a type which is an instance of the type of the function it denotes). In other terms, all λ-abstracted variables have monomorphic types (with respect to parametric polymorphism), which is why in ML auto-application $\lambda x.xx$ is not typeable.

The solution is well known and consists in introducing `let` bindings. This amounts to defining a new class of expression variables so that variables introduced by a `let` have polymorphic types, that is, types that have been generalized at the moment of the definition and can be instantiated in the body of the let. To sum up, λ-abstracted variables have monomorphic types, while `let`-bound variables (may) have polymorphic types and, thus, be given different types obtained by instantiation. For short we call the former λ-abstracted variables "monomorphic (expression) variables" and the latter `let`-bound variables "polymorphic (expression) variables".

In the explicitly-typed calculi of the previous sections we had just λ-abstracted variables. That these variables have monomorphic types is clearly witnessed by the fact that, operationally, $x\sigma_I$ is equivalent to (*ie*, reduces to) x. Clearly this property must not hold for polymorphic type variables since

$$\mathtt{let}\ x = (\lambda^{\alpha \to \alpha} y.y)\ \mathtt{in}\ (x[\{\alpha \to \alpha/\alpha\}])x \tag{23}$$

is, intuitively, well typed, while the same term obtained by replacing $x[\{\alpha \to \alpha/\alpha\}]$ with x is not (see the extension of the definition of relabeling for polymorphic variables later on).

To enable the definition of polymorphic functions to our calculus we add a `let` expression. To ease the presentation and to stress that the addition of `let`-bindings is a conservative extension of the previous system, we syntactically distinguish the current monomorphic variables (*ie*, those abstracted by a λ) from polymorphic variables by underlining the latter ones.

$$e ::= \cdots \mid \underline{x} \mid \mathtt{let}\ \underline{x} = e\ \mathtt{in}\ e$$

Reduction is as usual:

$$\mathtt{let}\ \underline{x} = v\ \mathtt{in}\ e \rightsquigarrow e\{v/\underline{x}\}$$

Relabeling is extended by the following definitions

$$\begin{array}{rcl} \underline{x}@[\sigma_j]_{j \in J} & \stackrel{\text{def}}{=} & \underline{x}[\sigma_j]_{j \in J} \\ (\mathtt{let}\ \underline{x} = e'\ \mathtt{in} e)@[\sigma_j]_{j \in J} & \stackrel{\text{def}}{=} & \mathtt{let}\ \underline{x} = (e'@[\sigma_j]_{j \in J})\ \mathtt{in} (e@[\sigma_j]_{j \in J}) \end{array}$$

and the (algorithmic) typing rule is as expected:

$$(let)\ \frac{\Delta\, ; \Gamma \vdash_{(\mathscr{A})} e_1 : t_1 \qquad \Delta\, ; \Gamma, (\underline{x} : t_1) \vdash_{(\mathscr{A})} e_2 : t_2}{\Delta\, ; \Gamma \vdash_{(\mathscr{A})} \mathtt{let}\ \underline{x} = e_1\ \mathtt{in}\ e_2 : t_2}$$

Type environments Γ now map also polymorphic expression variables into types. Notice that for a polymorphic expression variable $\underline{x}$ it is no longer true that $\mathsf{var}(\Gamma(\underline{x})) \subseteq \Delta$ (not adding $\mathsf{var}(\Gamma(\underline{x}))$ to Δ corresponds to generalizing the type of $\underline{x}$ before typing e_2 as in the GEN rule of the Damas-Milner algorithm: *cf.* [20]). As before we assume that polymorphic type variables of a `let`-expression (in particular those generalized for let-polymorphism) are distinct from monomorphic and polymorphic type variables of the context that the `let`-expression occurs in.

Likewise, environments now map both monomorphic and polymorphic expression variables into values so that the rules for evaluation in Section 5.2 are extended with the following ones:

$$\text{(PE-PVAR}_c\text{)}\ \frac{\mathscr{E}(\underline{x}) = c}{\sigma_I ; \mathscr{E} \vdash_{\mathsf{p}} \underline{x} \Downarrow c} \qquad \text{(PE-PVAR}_f\text{)}\ \frac{\mathscr{E}(\underline{x}) = \langle \lambda^t y.e, \mathscr{E}', \sigma_J \rangle}{\sigma_I ; \mathscr{E} \vdash_{\mathsf{p}} \underline{x} \Downarrow \langle \lambda^t y.e, \mathscr{E}', \sigma_I \circ \sigma_J \rangle}$$

$$\text{(PE-LET)}\ \frac{\sigma_I ; \mathscr{E} \vdash_{\mathsf{p}} e_1 \Downarrow v_0 \qquad \sigma_I ; \mathscr{E}, \underline{x} \mapsto v_0 \vdash_{\mathsf{p}} e_2 \Downarrow v}{\sigma_I ; \mathscr{E} \vdash_{\mathsf{p}} \mathtt{let}\ \underline{x} = e_1\ \mathtt{in}\ e_2 \Downarrow v}$$

To compile `let`-expressions we have to extend the intermediate language likewise: we will distinguish polymorphic expression variables by decorating them with sets of type-substitution formulæ Σ that apply to *that particular occurrence* of the variable. So we add to the productions of Section 5.3 the following ones:

$$e ::= \cdots \mid x_{\Sigma} \mid \mathtt{let}\ \underline{x} = e\ \mathtt{in}\ e$$

while the big-step semantics of the new added expressions is

$$\text{(OE-PVAR}_c\text{)}\quad \frac{\mathscr{E}(\underline{x}) = c}{\mathscr{E} \vdash_\circ x_\Sigma \Downarrow c} \qquad \text{(OE-PVAR}_f\text{)}\quad \frac{\mathscr{E}(\underline{x}) = \langle \lambda^t_{\Sigma'} y.e, \mathscr{E}' \rangle}{\mathscr{E} \vdash_\circ x_\Sigma \Downarrow \langle \lambda^t_{\mathtt{comp}(\Sigma,\Sigma')} y.e, \mathscr{E}' \rangle}$$

$$\text{(OE-LET)}\quad \frac{\mathscr{E} \vdash_\circ e_1 \Downarrow v_0 \qquad \mathscr{E}, \underline{x} \mapsto v_0 \vdash_\circ e_2 \Downarrow v}{\mathscr{E} \vdash_\circ \mathtt{let}\ \underline{x} = e_1\ \mathtt{in}\ e_2 \Downarrow v}$$

Notice how rule (OE-PVAR$_f$) uses the Σ decoration on the variable to construct the closure. The final step is the extension of the compiler for the newly added terms:

$$\begin{array}{rcl} [\![\underline{x}]\!]_\Sigma & = & x_\Sigma \\ [\![\mathtt{let}\ \underline{x} = e_1\ \mathtt{in}\ e_2]\!]_\Sigma & = & \mathtt{let}\ \underline{x} = [\![e_1]\!]_\Sigma\ \mathtt{in}\ [\![e_2]\!]_\Sigma \end{array}$$

As an example, the `let`-expression (23) is compiled into

$$\mathtt{let}\ \underline{x} = (\lambda^{\alpha\to\alpha} y.y)\ \mathtt{in}\ x_{[\{\alpha\to\alpha/\alpha\}]}\hat{x}$$

where the substitution $[\{\alpha\to\alpha/\alpha\}]$ that is applied to the leftmost occurrence of x is recorded in the variable and will be used to instantiate the closure associated with x by the environment; the rightmost occurrence of x is decorated by $\hat{\imath}$ and therefore the value bound to it will not be instantiated. Theorems 5.1, 5.2, and Corollary 5.3 hold also for these extensions (see Appendix D for the proofs).

In an actual programming language there will not be any syntactic distinction between the two kinds of expression variables and compilation can be optimized by transforming variables that are `let`-bound to monomorphic values into monomorphic variables. So, whenever e_1 has a monomorphic type t_1, the `let`-expression should be compiled as

$$[\![\mathtt{let}\ \underline{x} = e_1\ \mathtt{in}\ e_2]\!]_\Sigma \quad = \quad [\![(\lambda^{t_1\to t_2} x.e_2\{x/\underline{x}\})e_1]\!]_\Sigma$$

where t_2 is the type deduced for e_2 under the hypothesis that $\underline{x}$ has type t_1. Notice that this optimization is compositional.

6. Design choices and extensions

For the sake of concision we omitted two key features in the presentation: recursive functions and pairs. Recursive functions can be straightforwardly added with minor modifications. In particular, for recursive functions, whose syntax is $\mu f^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e$, it suffices to add in the type environment Γ the recursion variable f associated with the type obtained by applying the decoration to the interface, that is, $f : \wedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$: the reader can refer to Section 7.5 in [12] for a discussion on how and why recursion is restricted to functions.

The extension with product types, instead, is less straightforward but can be mostly done by using existing techniques. Syntactically, we add pairs (e, e) and projections $\pi_i e$ (for i=1, 2) to terms and the product type constructor $t\times t$ to types. Reduction semantics is standard: two notions of reduction $\pi_i(v_1, v_2) \rightsquigarrow v_i$ (for i=1, 2) plus the usual context reduction rules. Typing rules are standard, as well: a pair is typed by the product of the types of its components and if e is of type $t_1\times t_2$, then its i-th projection $\pi_i e$ has type t_i. The rule for pairs in the algorithmic system $\vdash_{\mathscr{A}}$ is the same as in the static semantics, while the rules for projections $\pi_i e$ become more difficult because the type inferred for e may not be of the form $t_1\times t_2$ but, in general, is (equivalent to) a union of intersections of types. We already met the latter problem for application (where the function type may be different from an arrow) and there we checked that the type deduced for the function in an application is a functional type (*ie*, a subtype of $\mathbb{0}\to\mathbb{1}$). Similarly, for products we must check that the type of e is a product type (*ie*, a subtype of $\mathbb{1}\times\mathbb{1}$). If the constraint is satisfied, then it is possible to define the type of the projection (in a way akin to the definition of the domain $\mathsf{dom}()$ for function types) using standard techniques of semantic subtyping (see Section 6.11 in [12]). This is explained in details in Appendix C.

Another concession to the sake of concision is the use of the relabeling operation '@' in the premises of both "abstr" rules in Figures 1 and 2. A slightly different but better formulation would have been to use as premises ... $\vdash_* e\sigma_j : s_i\sigma_j$ instead of ... $\vdash_* e@[\sigma_j] : s_i\sigma_j$, where $e\sigma$ denotes the application of a substitution σ to a term e and is roughly defined as the term obtained by applying σ to all interfaces in e and by composing σ within the outermost sets of type-substitutions in e. Both formulations are sound and the differences are really minimalist, but this second formulation rules out few anomalies of the current system. For instance, with the current formulation $\lambda^{\mathtt{Int}\to\mathtt{Int}}_D y.(\lambda^{\alpha\to\alpha}_{[]} x.42)[\{\mathtt{Int}/\alpha\}]y)$ is well-typed if and only if the decoration D is a non empty set of types-substitutions (whatever they are). Indeed a non empty D triggers the use of '@' in the premises of the "abstr" rule, and this "pushes" the type substitution $[\{\mathtt{Int}/\alpha\}]$ into the decoration of the body, thus making the body well typed (taken in isolation, $\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} x.42$ is well typed while $(\lambda^{\alpha\to\alpha}_{[]} x.42)[\{\mathtt{Int}/\alpha\}]$ is not). Although the second formulation rules out such kind of anomalies, we preferred to present the first one since it does not need the introduction of such technically-motivated new definitions.

For what concerns future extensions, in this work we dodged the problem of the negation of arrow types. Notice indeed that a value can have as type the negation of an arrow type just by subsumption. This implies that no λ-abstraction can have a negated arrow type. So while the type $\neg(\mathtt{Bool}\to\mathtt{Bool})$ can be inferred for, say, $(3, 42)$, it is not possible to infer it for $\lambda^{\mathtt{Int}\to\mathtt{Int}} x.(x+1)$. This problem was dealt in $\mathbb{C}$Duce by deducing for a λ-abstraction the type in its interface intersected with any negations of arrow types that did not make the type empty. Technically, this was dealt with "type schemas": a function such as $\lambda^{\mathtt{Int}\to\mathtt{Int}} x.x + 1$ has type schema $\{\{\mathtt{Int}\to\mathtt{Int}\}\}$, that is, it has every non empty type of the form $(\mathtt{Int}\to\mathtt{Int}) \wedge \bigwedge_{i\in I}\neg(s_i\to t_i)$ (thus, in particular, $(\mathtt{Int}\to\mathtt{Int}) \wedge \neg(\mathtt{Bool}\to\mathtt{Bool})$) [12]. In our context, however, the presence of type variables makes a definition such as that of schemas more difficult since a type schema should probably denote only types that are not empty for every possible instantiation of their variables (and it should probably be given at semantic level). Dealing with this aspect is mainly of theoretical interest (it allows to interpret types as sets of values, for instance in our case one could study as a possible interpretation $[\![t]\!] = \{v \mid\vdash v : s, s \sqsubseteq_\varnothing t\}$, where $\sqsubseteq_\Delta$ is defined in the companion paper) as witnessed by the fact that the $\mathbb{C}$Duce compiler does not use type schemas. We prefer to leave this study for future work.

For the semantics of the calculus we made few choices that restrict its generality. One of these, the use of a call-by-value reduction, is directly inherited from $\mathbb{C}$Duce and it is required to ensure subject reduction. If e is an expression of type $\mathtt{Int}\vee\mathtt{Bool}$, then the application $(\lambda^{(\mathtt{Int}\to\mathtt{Int}\times\mathtt{Int})\wedge(\mathtt{Bool}\to\mathtt{Bool}\times\mathtt{Bool})} x.(x, x))e$ has type $(\mathtt{Int}\times\mathtt{Int})\wedge(\mathtt{Bool}\times\mathtt{Bool})$. If we use call-by-name, then this redex reduces to (e, e) whose type $(\mathtt{Int}\vee\mathtt{Bool}\times\mathtt{Int}\vee\mathtt{Bool})$ is larger than the type of the redex. Although the use of call-by-name would not hinder the soundness of the type system (expressed in terms of progress) we preferred to ensure subject reduction since it greatly simplifies the theoretical development.

A second choice, to restrict type-cases to closed types, was made by practical considerations: using open types in a type-case would have been computationally prohibitive insofar as it demands to solve at run-time the problem whether for two given types s and t there exists a type-substitution σ such that $s\sigma \leq t\sigma$ (we study this problem, that we call the tallying problems, in the companion paper [5]). Our choice, instead, is compatible with the highly optimized (and provably optimal) pattern matching compilation technique of $\mathbb{C}$Duce. We leave for future work the study of type-cases on types with monomorphic variables (*ie*, those in Δ). This does not require dynamic type tallying resolution and would allow the

programmer to test capabilities of arguments bound to polymorphic type variables.

7. Related work

We focus on work related to this specific part of our work, namely, existing explicitly-typed calculi with intersection types, and functional languages to process XML data. Comparison with work on local type inference and type reconstruction is done in the second part of this work presented in the companion paper [5].

To compare the differences between the existing explicitly-typed calculi for intersection type systems, we discuss how the term of our daffy identity $(\lambda^{\alpha\to\alpha}_{[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]}x.(\lambda^{\alpha\to\alpha}y.x)x)$ is rendered.

In [23, 26], typing derivations are written as terms: different typed representatives of the same untyped term are joined together with an intersection $\wedge$. In such systems, the function in (7) relabeled with $[\{\texttt{Int}/\alpha\},\{\texttt{Bool}/\alpha\}]$ is written $(\lambda^{\texttt{Int}\to\texttt{Int}}x.(\lambda^{\texttt{Int}\to\texttt{Int}}y.x)x)\wedge(\lambda^{\texttt{Bool}\to\texttt{Bool}}x.(\lambda^{\texttt{Bool}\to\texttt{Bool}}y.x)x)$. Type checking verifies that both $\lambda^{\texttt{Int}\to\texttt{Int}}x.(\lambda^{\texttt{Int}\to\texttt{Int}}y.x)x$ and $\lambda^{\texttt{Bool}\to\texttt{Bool}}x.(\lambda^{\texttt{Bool}\to\texttt{Bool}}y.x)x$ are well typed separately, which generates two very similar typing derivations. The proposal of [16] follows the same idea, except that a separation is kept between the computational and the logical contents of terms. A term consists in the association of a *marked term* and a *proof term*. The marked term is just an untyped term where term variables are marked with integers. The proof term encodes the structure of the typing derivation and relates marks to types. The aforementioned example is written in this system as $(\lambda x:0.(\lambda y:1.x)x)@((\lambda 0^{\texttt{Int}}.(\lambda 1^{\texttt{Int}}.0)0)\wedge(\lambda 0^{\texttt{Bool}}.(\lambda 1^{\texttt{Bool}}.0)0))$. In general, different occurrences of a same mark can be paired with different types in the proof term. In [4], terms are duplicated (as in [23, 26]), but the type checking of terms does not generate copies of almost identical proofs. The type checking derivation for the term $((\lambda^{\texttt{Int}\to\texttt{Int}}x.(\lambda^{\texttt{Int}\to\texttt{Int}}y.x)x)\|\lambda^{\texttt{Bool}\to\texttt{Bool}}x.(\lambda^{\texttt{Bool}\to\texttt{Bool}}y.x)x)$ verifies in parallel that the two copies are well typed.

The duplication of terms and proofs makes the definition of beta reduction (and other transformations on terms) more difficult in the calculi presented so far, because it has to be performed in parallel on all the typed instances that correspond to the same untyped term. *Branching types* have been proposed in [27] to circumvent this issue. The idea is to represent different typing derivations for a same term into a compact piece of syntax. To this end, the branching type which corresponds to a given intersection type t records the *branching shape* (*ie*, the uses of the intersection introduction typing rule) of the typing derivation corresponding to t. For example, the type $(\texttt{Int}\to\texttt{Int})\wedge(\texttt{Bool}\to\texttt{Bool})$ has only two branches, which is represented in [27] by the branching shape $\mathsf{join}\{i{=}*,j{=}*\}$. Our running example is then written as $\Lambda\mathsf{join}\{i{=}*,j{=}*\}.\lambda x^{\{i=\texttt{Int},j=\texttt{Bool}\}}.(\lambda y^{\{i=\texttt{Int},j=\texttt{Bool}\}}.x)x$. Note that the lambda term itself is not copied, and no duplication of proofs happens during type checking either: the branches labeled i and j are verified in parallel.

In [9], the authors propose an expressive refinement type system with intersection, union, but also (a form of) dependent types, making possible to define, *eg*, the type of integer lists of length n, written $[\texttt{Int}]^n$. The variable n can be quantified over either universally or existentially (using respectively Π and Σ). Thanks to this it is possible to consider different instantiations of a dependent type and, thus to type our daffy function by different instances of $[\texttt{Int}]^n$ (rather than with any type, as for `Int` and `Bool` in our example). Type checking requires type annotations to be decidable: to check that $\lambda x.(\lambda y.x)x$ has type $\Pi n.(([\texttt{Int}]^{2n}\to[\texttt{Int}]^{2n})\wedge([\texttt{Int}]^{2n+1}\to[\texttt{Int}]^{2n+1}))$, the subterm $\lambda y.x$ has to be annotated. This problem is similar to finding appropriate annotations for the daffy function (7) in our language. In [9], terms are annotated with a list of typings: for example, $\lambda y.x$ can be annotated with $A=(x:[\texttt{Int}]^{2n}\vdash\mathbb{1}\to[\texttt{Int}]^{2n}, x:[\texttt{Int}]^{2n+1}\vdash\mathbb{1}\to[\texttt{Int}]^{2n+1})$, which says that if $x:[\texttt{Int}]^{2n}$, then $\lambda y.x$ has type $\mathbb{1}\to[\texttt{Int}]^{2n}$ (and similarly if $x:[\texttt{Int}]^{2n+1}$). The above annotation A is not sound because when checking that $\lambda x.(\lambda y.x:A)x$ has result type $\Pi n.(([\texttt{Int}]^{2n}\to[\texttt{Int}]^{2n})\wedge([\texttt{Int}]^{2n+1}\to[\texttt{Int}]^{2n+1}))$, one can see that the occurrences of n in A escape their scope: they should be bound by the quantifier Π in the result type. To fix this, typing environments in annotations are extended with universally quantified variables, that can be instantiated at type checking. For example, $\lambda y.x:(m:\texttt{Nat}, x:[\texttt{Int}]^m\vdash\mathbb{1}\to[\texttt{Int}]^m)$ means that $\lambda y.x$ has type $\mathbb{1}\to[\texttt{Int}]^m$, assuming $x:[\texttt{Int}]^m$, where m can be instantiated with any natural number. With this annotation, the daffy function can be checked against $\Pi n.(([\texttt{Int}]^{2n}\to[\texttt{Int}]^{2n})\wedge([\texttt{Int}]^{2n+1}\to[\texttt{Int}]^{2n+1}))$, by instantiating m with respectively $2n$ and $2n{+}1$. It is possible to find a similarity between the annotations of [9] (the lists of typings) and our annotations (*ie*, the combination of interface and decoration) although instantiation in the former is much more harnessed. There is however a fundamental difference between the two systems and it is that [9] does not include a type case. Because of that annotations need not to be propagated at run-time: in [9] they are just used statically to check soundness and then erased at run-time. Without type-cases we could do the same, but it is precisely the presence of type-cases that justifies our formalism.

For what concerns XML programming, let us cite polymorphic XDuce [13] and the work by Vouillon [25]. In both, pattern matching is designed so as not to break polymorphism, but both have to give up something: higher-order functions for [13] and intersection, negation, and local type inference in [25] (the type of function arguments must be explicitly given). Furthermore, Vouillon's work suffers from the original sin of starting from a subtyping relation that is given axiomatically by a deduction system. This makes the intuition underlying subtyping very difficult to grasp (at least, for us). Another route taken is the one of OCamlDuce [11], which juxtaposes OCaml and $\mathbb{C}$Duce's type systems in the same language, keeping them separated. This practical approach yields little theoretical problems but forces a value to be of one kind of type or another, preventing the programmer from writing polymorphic XML transformations. Lastly, XHaskell by Sulzmann *et al.* [24] mixes Haskell type classes with XDuce regular expression types but has two main drawbacks. First, every polymorphic variable must be annotated wherever it is instantiated with an XML type. Second, even without inference of explicit annotations (which they do not address), their system requires several restrictions to be decidable (while our system with explicit type-substitutions is decidable).

8. Conclusion

The work presented in this and in its companion paper [5] provides the theoretical basis and all the algorithmic tools needed to design and implement polymorphic functional languages for semi-structured data and, more generally, for functional languages with recursive types and set-theoretic unions, intersections, and negations. In particular, our results pave the way to the polymorphic extension of $\mathbb{C}$Duce [2] and to the definition of a real type system for XQuery 3.0 [10] (not just one in which all higher-order functions have type "`function()`"). Thanks to local type inference and type reconstruction defined in the second part of this work, these languages can have a syntax and semantics very close to those of OCaml or Haskell, but will include primitives (in particular, complex patterns) to exploit the great expressive power of full-fledged set-theoretic types.

Some problems are still open, notably the decidability of type-substitution inference defined in the second part of this work, but these are of theoretical nature and should not have any impact in practice (as a matter of facts people program in Java and Scala even

though the decidability of their type systems is still an open question). On the practical side, the most interesting directions of research is to couple the efficient compilation of the polymorphic calculus with techniques of static analysis that would perform partial evaluation of relabeling so as to improve the efficiency of typecase of functional values even in the rare cases in which it is more expensive than in the monomorphic version of $\mathbb{C}$Duce.

Acknowledgments. This work was partially supported by the ANR TYPEX project n. ANR-11-BS02-007. Zhiwu Xu was also partially supported by an Eiffel scholarship of French Ministry of Foreign Affairs and by the grant n. 61070038 of the National Natural Science Foundation of China.

References

[1] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03*. ACM Press, 2003.

[3] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. *XQuery 1.0: An XML Query Language*. W3C Working Draft, http://www.w3.org/TR/xquery/, May 2003.

[4] V. Bono, B. Venneri, and L. Bettini. A typed lambda calculus with intersection types. *Theor. Comput. Sci.*, 398(1-3):95–113, 2008.

[5] G. Castagna, K. Nguyễn, and Z. Xu. Polymorphic functions with set-theoretic types. Part 2: Local type inference and type reconstruction. Unpublished manuscript, available at http://hal.archives-ouvertes.fr/hal-00880744, November 2013.

[6] G. Castagna and Z. Xu. Set-theoretic Foundation of Parametric Polymorphism and Subtyping. In *ICFP '11*, 2011.

[7] J. Clark and M. Murata. Relax-NG, 2001. www.relaxng.org.

[8] M. Coppo, M. Dezani, and B. Venneri. Principal type schemes and lambda-calculus semantics. In *To H.B. Curry. Essays on Combinatory Logic, Lambda-calculus and Formalism*. Academic Press, 1980.

[9] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *POPL '04*. ACM Press, 2004.

[10] J. Robie et al. Xquery 3.0: An XML query language (working draft 2010/12/14), 2010. http://www.w3.org/TR/xquery-30/.

[11] A. Frisch. OCaml + XDuce. In *ICFP '06*, 2006.

[12] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *The Journal of ACM*, 55(4):1–64, 2008.

[13] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. *ACM TOPLAS*, 32(1):1–56, 2009.

[14] H. Hosoya and B.C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.

[15] C. Kirkegaard and A. Møller. XAct - XML transformations in Java. In *Programming Language Technologies for XML (PLAN-X)*, 2006.

[16] L. Liquori and S. Ronchi Della Rocca. Intersection-types à la Church. *Inf. Comput.*, 205(9):1371–1386, 2007.

[17] K. Zhuo Ming Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer, 2004.

[18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[19] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[20] F. Pottier and D. Rémy. The essence of ML type inference. In B.C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[21] J.C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.

[22] J.C. Reynolds. What do types mean?: from intrinsic to extrinsic semantics. In *Programming methodology*. Springer, 2003.

[23] S. Ronchi Della Rocca. Intersection typed lambda-calculus. *Electr. Notes Theor. Comput. Sci.*, 70(1):163–181, 2002.

[24] M. Sulzmann, K. Zhuo, and M. Lu. XHaskell - Adding Regular Expression Types to Haskell. In *IFL*, LNCS n. 5083. Springer, 2007.

[25] J. Vouillon. Polymorphic regular tree types and patterns. In *POPL '06*, pages 103–114, 2006.

[26] J.B. Wells, A. Dimock, R. Muller, and F.A. Turbak. A calculus with polymorphic and polyvariant flow types. *J. Funct. Program.*, 12(3):183–227, 2002.

[27] J.B. Wells and C. Haack. Branching types. In *ESOP '02*, volume 2305 of *LNCS*, pages 115–132. Springer, 2002.

[28] Z. Xu. *Parametric Polymorphism for XML Processing Languages*. PhD thesis, Université Paris Diderot, 2013. Available at http://tel.archives-ouvertes.fr/tel-00858744.

Appendix

A. Explicitly-Typed Calculus

In this section, we define our explicitly-typed λ-calculus with sets of type-substitutions that we outlined in Section 3.

A.1 Types

Definition A.1 (Types). *Let $\mathscr{V}$ be a countable set of type variables ranged over by Greek letters $\alpha,\beta,\gamma,\ldots$, and $\mathscr{B}$ a finite set of basic (or constant) types ranged over by b. A* type *is a term co-inductively produced by the following grammar*

$$\begin{array}{lrll} \textbf{\textit{Types}} & t ::= & \alpha & \textit{type variable} \\ & & \mid b & \textit{basic} \\ & & \mid t \times t & \textit{product} \\ & & \mid t \to t & \textit{arrow} \\ & & \mid t \vee t & \textit{union} \\ & & \mid \neg t & \textit{negation} \\ & & \mid \mathbb{0} & \textit{empty} \end{array}$$

that satisfies two additional requirements:

- *(regularity) the term must have a finite number of different sub-terms.*
- *(contractivity) every infinite branch must contain an infinite number of occurrences of atoms (*ie*, either a type variable or the immediate application of a type constructor: basic, product, arrow).*

We use $\mathscr{T}$ to denote the set of all types.

We write $t_1 \backslash t_2$, $t_1 \wedge t_2$, and $\mathbb{1}$ respectively as an abbreviation for $t_1 \wedge \neg t_2$, $\neg(\neg t_1 \vee \neg t_2)$, and $\neg\mathbb{0}$. The condition on infinite branches bars out ill-formed types such as $t = t \vee t$ (which does not carry any information about the set denoted by the type) or $t = \neg t$ (which cannot represent any set). It also ensures that the binary relation $\rhd \subseteq \mathscr{T}^2$ defined by $t_1 \vee t_2 \rhd t_i, \neg t \rhd t$ is Noetherian (that is, strongly normalizing). This gives an induction principle on $\mathscr{T}$ that we will use without any further explicit reference to the relation.

Since types are infinite, the accessory definitions on them will be given either by using memoization (*eg*, the definition of $\mathsf{var}()$, the variables occurring in a type: Definition A.2), by co-inductive techniques (*eg*, the definition or the application of type-substitutions: Definition A.5), or by induction on the relation $\rhd$, but only when induction does not traverse a type constructor (*eg*, the definition of $\mathsf{tlv}()$, the variables occurring at top-level of a type: Definition A.3).

Definition A.2 (Type variables). *Let var_0 and var_1 be two functions from $\mathscr{T} \times \mathscr{P}(\mathscr{T})$ to $\mathscr{P}(\mathscr{V})$ defined as:*

$$\begin{array}{rcl} var_0(t, \sqsupset) & = & \begin{cases} \emptyset & \textit{if } t \in \sqsupset \\ var_1(t, \sqsupset \cup \{t\}) & \textit{otherwise} \end{cases} \\ var_1(\alpha, \sqsupset) & = & \{\alpha\} \\ var_1(b, \sqsupset) & = & \emptyset \\ var_1(t_1 \times t_2, \sqsupset) & = & var_0(t_1, \sqsupset) \cup var_0(t_2, \sqsupset) \\ var_1(t_1 \to t_2, \sqsupset) & = & var_0(t_1, \sqsupset) \cup var_0(t_2, \sqsupset) \\ var_1(t_1 \vee t_2, \sqsupset) & = & var_1(t_1, \sqsupset) \cup var_1(t_2, \sqsupset) \\ var_1(\neg t_1, \sqsupset) & = & var_1(t_1, \sqsupset) \\ var_1(\mathbb{0}, \sqsupset) & = & \emptyset \end{array}$$

The set of type variables *occurring in a type t, written $\mathsf{var}(t)$, is defined as $var_0(t, \emptyset)$. A type t is said to be* ground *or* closed *if and only if $\mathsf{var}(t)$ is empty. We write $\mathscr{T}_0$ to denote the set of all the ground types.*

Definition A.3 (Top-level variables). *Let t be a type. The set $\mathsf{tlv}(t)$ of type variables that occur at top level in t, that is, all the variables of t that have at least one occurrence not under a constructor, is defined as:*

$$\begin{array}{rcl} \mathsf{tlv}(\alpha) & = & \{\alpha\} \\ \mathsf{tlv}(b) & = & \emptyset \\ \mathsf{tlv}(t_1 \times t_2) & = & \emptyset \\ \mathsf{tlv}(t_1 \to t_2) & = & \emptyset \\ \mathsf{tlv}(t_1 \vee t_2) & = & \mathsf{tlv}(t_1) \cup \mathsf{tlv}(t_2) \\ \mathsf{tlv}(\neg t_1) & = & \mathsf{tlv}(t_1) \\ \mathsf{tlv}(\mathbb{0}) & = & \emptyset \end{array}$$

Definition A.4 (Type substitution). *A* type-substitution *σ is a total mapping of type variables to types that is the identity everywhere but on a finite subset of $\mathscr{V}$, which is called the domain of σ and denoted by $\mathsf{dom}(\sigma)$. We use the notation $\{t_1/\alpha_1, \ldots, t_n/\alpha_n\}$ to denote the type-substitution that maps α_i to t_i for $i = 1..n$. Given a substitution σ, the* range *of σ is defined as the set of types $\mathsf{ran}(\sigma) = \{\sigma(\alpha) \mid \alpha \in \mathsf{dom}(\sigma)\}$, and the set of type variables occurring in the range is defined as $\mathsf{tvran}(\sigma) = \bigcup_{\alpha \in \mathsf{dom}(\sigma)} \mathsf{var}(\sigma(\alpha))$.*

Definition A.5. *Given a type $t \in \mathscr{T}$ and a type-substitution σ, the application of σ to t is co-inductively defined as follows:*

$$\begin{array}{rcll} b\sigma & = & b & \\ (t_1 \times t_2)\sigma & = & (t_1\sigma) \times (t_2\sigma) & \\ (t_1 \to t_2)\sigma & = & (t_1\sigma) \to (t_2\sigma) & \\ (t_1 \vee t_2)\sigma & = & (t_1\sigma) \vee (t_2\sigma) & \\ (\neg t)\sigma & = & \neg(t\sigma) & \\ \mathbb{0}\sigma & = & \mathbb{0} & \\ \alpha\sigma & = & \sigma(\alpha) & \text{if } \alpha \in \mathsf{dom}(\sigma) \\ \alpha\sigma & = & \alpha & \text{if } \alpha \notin \mathsf{dom}(\sigma) \end{array}$$

Definition A.6. *Let σ_1 and σ_2 be two substitutions such that $\mathsf{dom}(\sigma_1) \cap \mathsf{dom}(\sigma_2) = \emptyset$ ($\sigma_1 \sharp \sigma_2$ for short). Their* union *$\sigma_1 \cup \sigma_2$ is defined as*

$$(\sigma_1 \cup \sigma_2)(\alpha) = \begin{cases} \sigma_1(\alpha) & \alpha \in \mathsf{dom}(\sigma_1) \\ \sigma_2(\alpha) & \alpha \in \mathsf{dom}(\sigma_2) \\ \alpha & \text{otherwise} \end{cases}$$

A.2 Expressions

Definition A.7 (Expressions). *Let $\mathscr{C}$ be a set of constants ranged over by c and $\mathscr{X}$ a countable set of expression variables ranged over by $x,y,z,\ldots$. An* expression *e is a term inductively generated by the following grammar:*

Expressions	$e ::=$	c	*constant*
		$\mid x$	*expression variable*
		$\mid (e, e)$	*pair*
		$\mid \pi_i(e)$	*projection*$(i \in \{1,2\})$
		$\mid \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e$	*abstraction*
		$\mid e\ e$	*application*
		$\mid e{\in}t\ ?\ e : e$	*type case*
		$\mid e[\sigma_j]_{j\in J}$	*instantiation*

where t_i, s_i range over types, $t \in \mathscr{T}_0$ is a ground type and σ_j ranges over type-substitutions. We write $\mathscr{E}$ to denote the set of all expressions.

A λ-abstraction comes with a non-empty sequence of arrow types (called its *interface*) and a possibly empty set of type-substitutions (called its *decorations*). When the decoration is an empty set, we write $\lambda^{(\wedge_{i\in I} t_i \to s_i)} x.e$ for short.

Since expressions are inductively generated, the accessory definitions on them can be given by induction.

Given a set of type variables Δ and a set of type-substitutions $[\sigma_j]_{j\in J}$, for simplicity, we use the notation $\Delta[\sigma_j]_{j\in J}$ to denote the set of type variables occurring in the applications $\alpha\sigma_j$ for all $\alpha \in \Delta, j \in J$, that is:

$$\Delta[\sigma_j]_{j\in J} \stackrel{\text{def}}{=} \bigcup_{j\in J} \Big(\bigcup_{\alpha\in\Delta} \mathsf{var}(\sigma_j(\alpha)) \Big)$$

Definition A.8. *Let e be an expression. The set $\mathsf{fv}(e)$ of* free variables *of the expression e is defined by induction as:*

$$\begin{array}{rcl} \mathsf{fv}(x) & = & \{x\} \\ \mathsf{fv}(c) & = & \emptyset \\ \mathsf{fv}((e_1, e_2)) & = & \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \mathsf{fv}(\pi_i(e)) & = & \mathsf{fv}(e) \\ \mathsf{fv}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) & = & \mathsf{fv}(e) \setminus \{x\} \\ \mathsf{fv}(e_1\ e_2) & = & \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \mathsf{fv}(e{\in}t\ ?\ e_1 : e_2) & = & \mathsf{fv}(e) \cup \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\ \mathsf{fv}(e[\sigma_j]_{j\in J}) & = & \mathsf{fv}(e) \end{array}$$

The set $\mathsf{bv}(e)$ of bound variables *of the expression e is defined by induction as:*

$$\begin{array}{rcl} \mathsf{bv}(x) & = & \emptyset \\ \mathsf{bv}(c) & = & \emptyset \\ \mathsf{bv}((e_1, e_2)) & = & \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) \\ \mathsf{bv}(\pi_i(e)) & = & \mathsf{bv}(e) \\ \mathsf{bv}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) & = & \mathsf{bv}(e) \cup \{x\} \\ \mathsf{bv}(e_1\ e_2) & = & \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) \\ \mathsf{bv}(e{\in}t\ ?\ e_1 : e_2) & = & \mathsf{bv}(e) \cup \mathsf{bv}(e_1) \cup \mathsf{bv}(e_2) \\ \mathsf{bv}(e[\sigma_j]_{j\in J}) & = & \mathsf{bv}(e) \end{array}$$

The set $\mathsf{tv}(e)$ *of* type variables *occurring in* e *is defined by induction as:*

$$\begin{array}{rcl}
\mathsf{tv}(x) & = & \emptyset \\
\mathsf{tv}(c) & = & \emptyset \\
\mathsf{tv}((e_1, e_2)) & = & \mathsf{tv}(e_1) \cup \mathsf{tv}(e_2) \\
\mathsf{tv}(\pi_i(e)) & = & \mathsf{tv}(e) \\
\mathsf{tv}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) & = & \mathsf{tv}(e[\sigma_j]_{j\in J}) \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \\
\mathsf{tv}(e_1\ e_2) & = & \mathsf{tv}(e_1) \cup \mathsf{tv}(e_2) \\
\mathsf{tv}(e{\in}t\,?\,e_1 : e_2) & = & \mathsf{tv}(e) \cup \mathsf{tv}(e_1) \cup \mathsf{tv}(e_2) \\
\mathsf{tv}(e[\sigma_j]_{j\in J}) & = & (\mathsf{tv}(e))[\sigma_j]_{j\in J}
\end{array}$$

An expression e *is* closed *if* $\mathsf{fv}(e)$ *is empty.*

Note that the set of type variables in $e[\sigma_j]_{j\in J}$ is the set returned by the "application" of $[\sigma_j]_{j\in J}$ to the set $\mathsf{tv}(e)$.

Next, we define expression substitutions. Recall that, as stated in Section 3.1, when substituting an expression e for a variable y in an expression e', we suppose the polymorphic type variables of e' to be distinct from the monomorphic and polymorphic type variables of e to avoid unwanted captures. In the discussion in Section 3.1, the definitions of the notions of polymorphic and monomorphic variables remain informal. To make them more formal, we would have to distinguish between the two by carrying around a set of type variables Δ which would contain the monomorphic variables that cannot be α-converted. Then all definitions (such as expression substitutions, for example) would have to be parameterized with Δ, making the definitions and technical developments difficult to read just because of α-conversion. Therefore, for the sake of readability, we decided to keep the distinction between polymorphic and monomorphic variables informal.

Definition A.9 (Expression substitution). *An* expression substitution ϱ *is a total mapping of expression variables to expressions that is the identity everywhere but on a finite subset of* $\mathscr{X}$*, which is called the domain of* ϱ *and denoted by* $\mathsf{dom}(\varrho)$*. We use the notation* $\{e_1/x_1, \ldots, e_n/x_n\}$ *to denote the expression substitution that maps* x_i *into* e_i *for* $i = 1..n$*.*

The definitions of free variables, bound variables, and type variables are extended to expression substitutions as follows.

$$\mathsf{fv}(\varrho) = \bigcup_{x\in\mathsf{dom}(\varrho)} \mathsf{fv}(\varrho(x)), \quad \mathsf{bv}(\varrho) = \bigcup_{x\in\mathsf{dom}(\varrho)} \mathsf{bv}(\varrho(x)), \quad \mathsf{tv}(\varrho) = \bigcup_{x\in\mathsf{dom}(\varrho)} \mathsf{tv}(\varrho(x))$$

Next, we define the application of an expression substitution ϱ to an expression e. To avoid unwanted captures, we remind that we assume that the bound variables of e do not occur in the domain of ϱ and that the polymorphic type variables of e are distinct from the type variables occurring in ϱ (using α-conversion if necessary).

Definition A.10. *Given an expression* $e \in \mathscr{E}$ *and an expression substitution* ϱ*, the application of* ϱ *to* e *is defined as follows:*

$$\begin{array}{rcll}
c\varrho & = & c & \\
(e_1, e_2)\varrho & = & (e_1\varrho, e_2\varrho) & \\
(\pi_i(e))\varrho & = & \pi_i(e\varrho) & \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e)\varrho & = & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.(e\varrho) & \\
(e_1\ e_2)\varrho & = & (e_1\varrho)\ (e_2\varrho) & \\
(e{\in}t\,?\,e_1 : e_2)\varrho & = & e\varrho{\in}t\,?\,e_1\varrho : e_2\varrho & \\
x\varrho & = & \varrho(x) & \text{if } x \in \mathsf{dom}(\varrho) \\
x\varrho & = & x & \text{if } x \notin \mathsf{dom}(\varrho) \\
(e[\sigma_j]_{j\in J})\varrho & = & (e\varrho)[\sigma_j]_{j\in J} &
\end{array}$$

In the case for instantiation $(e[\sigma_j]_{j\in J})\varrho$, the σ_j operate on the polymorphic type variables, which we assume distinct from the variables in ϱ (using α-conversion if necessary). As a result, we have $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$. Similarly, in the abstraction case, we have $x \notin \mathsf{dom}(\varrho)$.

Next, we define the relabeling of an expression e with a set of type-substitutions $[\sigma_j]_{j\in J}$, which consists in propagating the σ_j to the λ-abstractions in e if needed. We suppose that the polymorphic type variables in e are distinct from the type variables in the range of σ_j (this is always possible using α-conversion).

Definition A.11 (Relabeling). *Given an expression* $e \in \mathscr{E}$ *and a set of type-substitutions* $[\sigma_j]_{j\in J}$*, we define the* relabeling *of* e *with* $[\sigma_j]_{j\in J}$*, written* $e@[\sigma_j]_{j\in J}$*, as* e *if* $\mathsf{tv}(e) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$*, and otherwise as follows:*

$$\begin{array}{rcl}
(e_1, e_2)@[\sigma_j]_{j\in J} & = & (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J}) \\
(\pi_i(e))@[\sigma_j]_{j\in J} & = & \pi_i(e@[\sigma_j]_{j\in J}) \\
(e_1\ e_2)@[\sigma_j]_{j\in J} & = & (e_1@[\sigma_j]_{j\in J})\ (e_2@[\sigma_j]_{j\in J}) \\
(e{\in}t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J} & = & e@[\sigma_j]_{j\in J}{\in}t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J} \\
(e[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J} & = & e@([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) \\
(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e)@[\sigma_j]_{j\in J} & = & \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e
\end{array}$$

The substitutions are not propagated if they do not affect the variables of e (i.e., if $\mathsf{tv}(e) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$). In particular, constants and variables are left unchanged, as they do not contain any type variable. Suppose now that $\mathsf{tv}(e) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) \neq \emptyset$. In the abstraction case, the propagated substitutions are composed with the decorations of the abstraction, without propagating them further down in the body. Propagation in the body occurs, whenever is needed, that is, during either reduction (see (*Rappl*) in Section A.4) or type-checking (see (*abstr*) in Section A.3). In the instantiation case $e[\sigma_i]_{i\in I}$, we propagate the result of the composition of $[\sigma_i]_{i\in I}$ with $[\sigma_j]_{j\in J}$ in e. The remaining cases are simple inductive cases. Finally notice that in a type case $e\in t\,?\,e_1 : e_2$, we do not apply $[\sigma_j]_{j\in J}$ to t, simply because t is ground.

A.3 Type System

Because of the type directed nature of our calculus (*ie*, the presence of the type-case expression), its dynamic semantics is defined only for well-typed expressions. Therefore, we introduce the type system before giving the reduction rules.

Definition A.12 (Typing environment). *A* typing environment Γ *is a finite mapping from expression variables $\mathscr{X}$ to types $\mathscr{T}$, and written as a finite set of pairs $\{(x_1 : t_1), \ldots, (x_n : t_n)\}$. The set of expression variables defined in Γ is called the domain of Γ, denoted by $\mathsf{dom}(\Gamma)$. The set of type variables occurring in Γ, that is, $\bigcup_{(x:t)\in\Gamma} \mathsf{var}(t)$, is denoted by $\mathsf{var}(\Gamma)$. If Γ is a type environment, then $\Gamma, (x : t)$ is the type environment defined as*

$$(\Gamma, (x : t))(y) = \begin{cases} t & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

We extend the definition of type-substitution application to type environments by applying the type-substitution to each type in the type environment as follows:

$$\Gamma\sigma = \{(x : t\sigma) \mid (x : t) \in \Gamma\}$$

The typing judgment for expressions has the form $\Delta \fatsemi \Gamma \vdash e : t$, which states that under the set Δ of (monomorphic) type variables and the typing environment Γ the expression e has type t. When Δ and Γ are both empty, we write $\vdash e : t$ for short. We assume that there is a basic type b_c for each constant c. We write $\sigma \sharp \Delta$ as abbreviation for $\mathsf{dom}(\sigma) \cap \Delta = \emptyset$. The typing rules are given in Figure 3 which are the same as in Section 3 except for the rules for products.

$$\frac{}{\Delta \fatsemi \Gamma \vdash c : b_c}\ (\textit{const}) \qquad \frac{}{\Delta \fatsemi \Gamma \vdash x : \Gamma(x)}\ (\textit{var}) \qquad \frac{\Delta \fatsemi \Gamma \vdash e_1 : t_1 \qquad \Delta \fatsemi \Gamma \vdash e_2 : t_2}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (\textit{pair})$$

$$\frac{\Delta \fatsemi \Gamma \vdash e : t_1 \times t_2}{\Delta \fatsemi \Gamma \vdash \pi_i(e) : t_i}\ (\textit{proj}) \qquad \frac{\Delta \fatsemi \Gamma \vdash e_1 : t_1 \to t_2 \qquad \Delta \fatsemi \Gamma \vdash e_2 : t_1}{\Delta \fatsemi \Gamma \vdash e_1 e_2 : t_2}\ (\textit{appl})$$

$$\frac{\begin{array}{l}\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \\ \forall i \in I, j \in J.\ \Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j\end{array}}{\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ (\textit{abstr})$$

$$\frac{\Delta \fatsemi \Gamma \vdash e : t' \qquad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \Delta \fatsemi \Gamma \vdash e_1 : s \\ t' \not\leq t & \Rightarrow \quad \Delta \fatsemi \Gamma \vdash e_2 : s \end{cases}}{\Delta \fatsemi \Gamma \vdash (e\in t\,?\,e_1 : e_2) : s}\ (\textit{case}) \qquad \frac{\Delta \fatsemi \Gamma \vdash e : t \qquad \sigma \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e[\sigma] : t\sigma}\ (\textit{inst})$$

$$\frac{\forall j \in J.\ \Delta \fatsemi \Gamma \vdash e[\sigma_j] : t_j \qquad |J| > 1}{\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (\textit{inter}) \qquad \frac{\Delta \fatsemi \Gamma \vdash e : s \qquad s \leq t}{\Delta \fatsemi \Gamma \vdash e : t}\ (\textit{subsum})$$

Figure 3. Typing rules

A.4 Operational Semantics

Definition A.13 (Values). *An expression e is a* value *if it is closed, well-typed (*ie, $\vdash e : t$ *for some type t), and produced by the following grammar:*

$$\textbf{\textit{Values}} \qquad v ::= c \mid (v, v) \mid \lambda^{(\wedge_{i\in I} t_i\to s_i)}_{[\sigma_j]_{j\in J}} x.e$$

We write $\mathscr{V}$ to denote the set of all values.

Definition A.14 (Context). *Let the symbol $[\_]$ denote a hole. A* context *$C[\_]$ is an expression with a hole:*

$$
\begin{array}{llll}
\textbf{\textit{Contexts}} & C[\_] & ::= & [\_] \\
 & & | & (C[\_], e) \mid (e, C[\_]) \\
 & & | & C[\_]\ e \mid e\ C[\_] \\
 & & | & C[\_] \in t\,?\,e : e \mid e \in t\,?\,C[\_] : e \mid e \in t\,?\,e : C[\_] \\
 & & | & \pi_i(C[\_])
\end{array}
$$

An evaluation context *$E[\_]$ is a context that implements outermost leftmost reduction:*

$$
\begin{array}{llll}
\textbf{\textit{Evaluation Contexts}} & E[\_] & ::= & [\_] \\
 & & | & (E[\_], e) \mid (v, E[\_]) \\
 & & | & E[\_]\ e \mid v\ E[\_] \\
 & & | & E[\_] \in t\,?\,e : e \\
 & & | & \pi_i(E[\_])
\end{array}
$$

We use $C[e]$ and $E[e]$ to denote the expressions obtained by replacing e for the hole in $C[\_]$ and $E[\_]$, respectively.

We define a small-step call-by-value operational semantics for the calculus. The semantics is given by the relation $\rightsquigarrow$, which is shown in Figure 4. There are four notions of reduction: one for projections, one for applications, one for type cases, and one for instantiations, plus context closure. Henceforth we will establish all the properties for the reduction using generic contexts but, of course, these holds also when the more restrictive evaluation contexts are used.

Notions of reduction:

(*Rproj*) $\pi_i(v_1, v_2) \rightsquigarrow v_i$

(*Rappl*) $(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e')v \rightsquigarrow (e'@[\sigma_j]_{j\in P})\{v/x\}$
where $P = \{j \in J \mid \exists i \in I.\ \vdash v : t_i\sigma_j\}$

(*Rcase*) $(v \in t\,?\,e_1 : e_2) \rightsquigarrow \begin{cases} e_1 & \text{if } \vdash v : t \\ e_2 & \text{otherwise} \end{cases}$

(*Rinst*) $e[\sigma_j]_{j\in J} \rightsquigarrow e@[\sigma_j]_{j\in J}$

Context closure:

(*Rctx*) $$\frac{e \rightsquigarrow e'}{C[e] \rightsquigarrow C[e']}$$

Figure 4. Operational semantics of the calculus

The (*Rproj*) rule is the standard projection rule while the other notions of reduction have already been explained in Section 3.3.

We used a call-by-value semantics for application to ensure the type soundness property: subject reduction (or type preservation) and progress (closed and well-typed expressions which are not values can be reduced), which are discussed in Section B.2. To understand why, consider each basic reduction rule in turn.

The requirement that the argument of a projection must be a value is imposed to ensure that the property of subject reduction holds. Consider the expression $e = \pi_1(e_1, e_2)$ where e_1 is an expression of type t_1 (different from $\mathbb{0}$) and e_2 is a (diverging) expression of type $\mathbb{0}$. Clearly, the type system assigns the type $t_1 \times \mathbb{0}$ to (e_1, e_2). In our system, a product type with an empty component is itself empty, and thus e has type $\mathbb{0}$. Therefore the type of the projection as well has type $\mathbb{0}$ (since $\mathbb{0} \leq \mathbb{0} \times \mathbb{0}$, then by subsumption $(e_1, e_2) : \mathbb{0} \times \mathbb{0}$ and the result follows from the (*proj*) typing rule). If it were possible to reduce a projection when the argument is not a value, then e could be reduced to e_1, which has type t_1: type preservation would be violated.

Likewise, the reduction rule for applications requires the argument to be a value. Let us consider the application $(\lambda^{(t\to t\times t)\wedge(s\to s\times s)}x.(x,x))(e)$, where $\vdash e : t \vee s$. The type system assigns to the abstraction the type $(t \to t \times t) \wedge (s \to s \times s)$, which is a subtype of $(t \vee s) \to ((t \times t) \vee (s \times s))$. By subsumption, the abstraction has type $(t \vee s) \to ((t \times t) \vee (s \times s))$, and thus, the application has type $(t \times t) \vee (s \times s)$. If the semantics permits to reduce an application when the argument is not a value, then this application could be reduced to the expression (e, e), which has type $(t \vee s) \times (t \vee s)$ but not $(t \times t) \vee (s \times s)$.

Finally, if we allowed $(e \in t \,?\, e_1 : e_2)$ to reduce to e_1 when $\vdash e : t$ but e is not a value, we could break type preservation. For example, assume that $\vdash e : \mathbb{0}$. Then the type system would not check anything about the branches e_1 and e_2 (see the typing rule (*case*) in Figure 3) and so e_1 could be ill-typed.

Notice that in all these cases the usage of values ensures subject reduction but it is not a necessary condition: in some cases weaker constraints could be used. For instance, in order to check whether an expression is a list of integers, in general it is not necessary to fully evaluate the whole list: the head and the type of the tail are all that is needed. Studying weaker conditions for the reduction rules is an interesting topic, which we leave for future work, in particular, in the view of adapting our framework to lazy languages.

B. Properties of the Type System

In this section we present some properties of our type system. First, we study its syntactic meta-theory: in particular, we prove admissibility of the intersection rule, a generation lemma for values, and that substitutions preserve typing. These properties are needed to prove *soundness*, the fundamental property which links every type system of a calculus with its operational counterpart: well-typed expressions do not go wrong [18]. Next, we prove that the explicitly-typed calculus is able to derive the same typing judgments as the BCD intersection type system defined by Barendregt, Coppo, and Dezani [1]. Finally, we prove that the expressions of the form $e[\sigma_j]_{j \in J}$ are redundant insofar as their presence in the calculus does not increase its expressive power.

B.1 Syntactic meta-theory

Lemma B.1. *If $\Delta \fatsemi \Gamma \vdash e : t$ and $\mathsf{var}(\Gamma) \subseteq \Delta$, then $\mathsf{var}(\Gamma') \subseteq \Delta'$ holds for every judgment $\Delta' \fatsemi \Gamma' \vdash e' : t'$ in the derivation of $\Delta \fatsemi \Gamma \vdash e : t$.*

Proof. By induction on the derivation of $\Delta \fatsemi \Gamma \vdash e : t$. □

Lemma B.2 (Admissibility of intersection introduction). *Let e be an expression. If $\Delta \fatsemi \Gamma \vdash e : t$ and $\Delta \fatsemi \Gamma \vdash e : t'$, then $\Delta \fatsemi \Gamma \vdash e : t \wedge t'$.*

Proof. The proof proceeds by induction on the two typing derivations. First, assume that these two derivations end with an instance of the same rule corresponding to the top-level constructor of e.

(*const*)**:** both derivations end with an instance of (*const*):

$$\frac{}{\Delta \fatsemi \Gamma \vdash c : b_c}\ (const) \qquad \frac{}{\Delta \fatsemi \Gamma \vdash c : b_c}\ (const)$$

Trivially, we have $b_c \wedge b_c \simeq b_c$, by subsumption, the result follows.

(*var*)**:** both derivations end with an instance of (*var*):

$$\frac{}{\Delta \fatsemi \Gamma \vdash x : \Gamma(x)}\ (var) \qquad \frac{}{\Delta \fatsemi \Gamma \vdash x : \Gamma(x)}\ (var)$$

Trivially, we have $\Gamma(x) \wedge \Gamma(x) \simeq \Gamma(x)$, by subsumption, the result follows.

(*pair*)**:** both derivations end with an instance of (*pair*):

$$\frac{\dots \quad \dots}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : (t_1 \times t_2)}\ (pair) \qquad \frac{\dots \quad \dots}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : (t_1' \times t_2')}\ (pair)$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e_i : (t_i \wedge t_i')$. Then the rule (*pair*) gives us $\Delta \fatsemi \Gamma \vdash (e_1, e_2) : (t_1 \wedge t_1') \times (t_2 \wedge t_2')$. Moreover, because intersection distributes over products, we have $(t_1 \wedge t_1') \times (t_2 \wedge t_2') \simeq (t_1 \times t_2) \wedge (t_1' \times t_2')$. Then by (*subsum*), we have $\Delta \fatsemi \Gamma \vdash (e_1, e_2) : (t_1 \times t_2) \wedge (t_1' \times t_2')$.

(*proj*)**:** both derivations end with an instance of (*proj*):

$$\frac{\dots}{\Delta \fatsemi \Gamma \vdash \pi_i(e') : t_i}\ (proj) \qquad \frac{\dots}{\Delta \fatsemi \Gamma \vdash \pi_i(e') : t_i'}\ (proj)$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e' : (t_1 \times t_2) \wedge (t_1' \times t_2')$. Since $(t_1 \wedge t_1') \times (t_2 \wedge t_2') \simeq (t_1 \times t_2) \wedge (t_1' \times t_2')$ (see the case of (*pair*)), by (*subsum*), we have $\Delta \fatsemi \Gamma \vdash e' : (t_1 \wedge t_1') \times (t_2 \wedge t_2')$. Then the rule (*proj*) gives us $\Delta \fatsemi \Gamma \vdash \pi_i(e') : t_i \wedge t_i'$.

(*appl*)**:** both derivations end with an instance of (*appl*):

$$\frac{\dots \quad \dots}{\Delta \fatsemi \Gamma \vdash e_1\ e_2 : t_2}\ (appl) \qquad \frac{\dots \quad \dots}{\Delta \fatsemi \Gamma \vdash e_1\ e_2 : t_2'}\ (appl)$$

By induction, we have $\Delta \fatsemi \Gamma \vdash e_1 : (t_1 \to t_2) \wedge (t_1' \to t_2')$ and $\Delta \fatsemi \Gamma \vdash e_2 : t_1 \wedge t_1'$. Because intersection distributes over arrows, we have $(t_1 \to t_2) \wedge (t_1' \to t_2') \leq (t_1 \wedge t_1') \to (t_2 \wedge t_2')$. Then by the rule (*subsum*), we get $\Delta \fatsemi \Gamma \vdash e_1 : (t_1 \wedge t_1') \to (t_2 \wedge t_2')$. Finally, by applying (*appl*), we get $\Delta \fatsemi \Gamma \vdash e_1\ e_2 : t_2 \wedge t_2'$ as expected.

(*abstr*)**:** both derivations end with an instance of (*abstr*):

$$\dfrac{\begin{array}{l}\forall i \in I, j \in J.\ \dfrac{\cdots}{\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j)\end{array}}{\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}$$

$$\dfrac{\begin{array}{l}\forall i \in I, j \in J.\ \dfrac{\cdots}{\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j)\end{array}}{\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}$$

It is clear that

$$(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \wedge (\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j) \simeq \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$$

By subsumption, the result follows.

(*case*)**:** both derivations end with an instance of (*case*):

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e_0 : t_0} \quad \begin{cases} t_0 \not\leq \neg t & \Rightarrow \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1 : s} \\ t_0 \not\leq t & \Rightarrow \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_2 : s} \end{cases}}{\Delta \wr \Gamma \vdash (e_0 \in t\,?\,e_1 : e_2) : s}\ (case)$$

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e_0 : t'_0} \quad \begin{cases} t'_0 \not\leq \neg t & \Rightarrow \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1 : s'} \\ t'_0 \not\leq t & \Rightarrow \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_2 : s'} \end{cases}}{\Delta \wr \Gamma \vdash (e_0 \in t\,?\,e_1 : e_2) : s'}\ (case)$$

By induction, we have $\Delta \wr \Gamma \vdash e_0 : t_0 \wedge t'_0$. Suppose $t_0 \wedge t'_0 \not\leq \neg t$; then $t_0 \not\leq \neg t$ and $t'_0 \not\leq \neg t$. Consequently, the branch e_1 has been type-checked in both cases, and we have $\Delta \wr \Gamma \vdash e_1 : s \wedge s'$ by the induction hypothesis. Similarly, if $t_0 \wedge t'_0 \not\leq t$, then we have $\Delta \wr \Gamma \vdash e_2 : s \wedge s'$. Consequently, we have $\Delta \wr \Gamma \vdash (e_0 \in t\,?\,e_1 : e_2) : s \wedge s'$ by the rule (*case*).

(*inst*)**:** both derivations end with an instance of (*inst*):

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : t} \quad \sigma \sharp \Delta}{\Delta \wr \Gamma \vdash e'[\sigma] : t\sigma}\ (inst) \qquad \dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : t'} \quad \sigma \sharp \Delta}{\Delta \wr \Gamma \vdash e'[\sigma] : t'\sigma}\ (inst)$$

By induction, we have $\Delta \wr \Gamma \vdash e' : t \wedge t'$. Since $\sigma \sharp \Delta$, the rule (*inst*) gives us $\Delta \wr \Gamma \vdash e'[\sigma] : (t \wedge t')\sigma$, that is $\Delta \wr \Gamma \vdash e'[\sigma] : (t\sigma) \wedge (t'\sigma)$.

(*inter*)**:** both derivations end with an instance of (*inter*):

$$\dfrac{\forall j \in J.\ \dfrac{\cdots}{\Delta \wr \Gamma \vdash e'[\sigma_j] : t_j}}{\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (inter) \qquad \dfrac{\forall j \in J.\ \dfrac{\cdots}{\Delta \wr \Gamma \vdash e'[\sigma_j] : t'_j}}{\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'_j}\ (inter)$$

where $|J| > 1$. By induction, we have $\Delta \wr \Gamma \vdash e'[\sigma_j] : t_j \wedge t'_j$ for all $j \in J$. Then the rule (*inter*) gives us $\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(t_j \wedge t'_j)$, that is, $\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : (\bigwedge_{j\in J} t_j) \wedge (\bigwedge_{j\in J} t'_j)$.

Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), for instance,

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : s} \quad s \leq t}{\Delta \wr \Gamma \vdash e' : t}\ (subsum) \qquad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : t'}$$

By induction, we have $\Delta \wr \Gamma \vdash e' : s \wedge t'$. Since $s \leq t$, we have $s \wedge t' \leq t \wedge t'$. Then the rule (*subsum*) gives us $\Delta \wr \Gamma \vdash e' : t \wedge t'$ as expected. □

Lemma B.3 (**Generation for values**). *Let v be a value. Then*

1. *If $\Delta \wr \Gamma \vdash v : b$, then v is a constant c and $b_c \leq b$.*
2. *If $\Delta \wr \Gamma \vdash v : t_1 \times t_2$, then v has the form of (v_1, v_2) with $\Delta \wr \Gamma \vdash v_i : t_i$.*
3. *If $\Delta \wr \Gamma \vdash v : t \to s$, then v has the form of $\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$ with $\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j) \leq t \to s$.*

Proof. By a simple examination of the rules it is easy to see that a derivation for $\Delta \wr \Gamma \vdash v : t$ is always formed by an instance of the rule corresponding to the kind of v (*i.e.*, (*const*) for constants, (*pair*) for pairs, and (*abstr*) for abstractions), followed by zero or more instances of (*subsum*). By induction on the depth of the derivation it is then easy to prove that if $\Delta \wr \Gamma \vdash v : t$ is derivable, then $t \not\simeq \mathbb{0}$. The lemma then follows

by induction on the number of the instances of the subsumption rule that end the derivation of $\Delta \wr \Gamma \vdash v : t$. The base cases are straightforward, while the inductive cases are:

$\underline{\Delta \wr \Gamma \vdash v : b}$**:** v is by induction a constant c such that $b_c \leq b$.

$\underline{\Delta \wr \Gamma \vdash v : t_1 \times t_2}$**:** v is by induction a pair (v_1, v_2) and t' is form of $(t'_1 \times t'_2)$ such that $\Delta \wr \Gamma \vdash v_i : t'_i$. Here we use the fact that the type of a value cannot be $\mathbb{0}$: since $\mathbb{0} \not\simeq (t'_1 \times t'_2) \leq (t_1 \times t_2)$, then we have $t'_i \leq t_i$. Finally, by (*subsum*), we have $\Delta \wr \Gamma \vdash v_i : t_i$.

$\underline{\Delta \wr \Gamma \vdash v : t \to s}$**:** v is by induction an abstraction $\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$ such that $\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j) \leq t \to s$.

□

Lemma B.4. *Let e be an expression and $[\sigma_j]_{j\in J}, [\sigma_k]_{k\in K}$ two sets of type substitutions. Then*

$$(e@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} = e@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})$$

Proof. By induction on the structure of e.

$\underline{e = c}$**:**

$$\begin{array}{rcl}(c@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& c@[\sigma_k]_{k\in K}\\ &=& c\\ &=& c@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{array}$$

$\underline{e = x}$**:**

$$\begin{array}{rcl}(x@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& x@[\sigma_k]_{k\in K}\\ &=& x\\ &=& x@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{array}$$

$\underline{e = (e_1, e_2)}$**:**

$$\begin{array}{rcll}((e_1, e_2)@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K}\\ &=& ((e_1@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K}, (e_2@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K}) & \text{(by induction)}\\ &=& (e_1@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}), e_2@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}))\\ &=& (e_1, e_2)@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{array}$$

$\underline{e = \pi_i(e')}$**:**

$$\begin{array}{rcll}(\pi_i(e')@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& (\pi_i(e'@[\sigma_j]_{j\in J}))@[\sigma_k]_{k\in K}\\ &=& \pi_i((e'@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K}) & \text{(by induction)}\\ &=& \pi_i(e'@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}))\\ &=& \pi_i(e')@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{array}$$

$\underline{e = e_1 e_2}$**:**

$$\begin{array}{rcll}((e_1 e_2)@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& ((e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}))@[\sigma_k]_{k\in K}\\ &=& ((e_1@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K})((e_2@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K}) & \text{(by induction)}\\ &=& (e_1@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}))(e_2@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J}))\\ &=& (e_1 e_2)@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{array}$$

$\underline{e = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_{j'}]_{j'\in J'}} x.e'}$**:**

$$\begin{array}{rcl}((\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_{j'}]_{j'\in J'}} x.e')@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& (\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J} \circ [\sigma_{j'}]_{j'\in J'}} x.e')@[\sigma_k]_{k\in K}\\ &=& (\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J} \circ [\sigma_{j'}]_{j'\in J'}} x.e')\\ &=& (\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_{j'}]_{j'\in J'}} x.e')@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{array}$$

$\underline{e = e_0 \in t \,?\, e_1 : e_2}$**:** similar to $e = (e_1, e_2)$

$\underline{e = e'[\sigma_i]_{i\in I}}$**:**

$$\begin{array}{rcll}((e'[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J})@[\sigma_k]_{k\in K} &=& (e'@([\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}))@[\sigma_k]_{k\in K}\\ &=& e'@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J} \circ [\sigma_i]_{i\in I}) & \text{(by induction)}\\ &=& (e'[\sigma_i]_{i\in I})@([\sigma_k]_{k\in K} \circ [\sigma_j]_{j\in J})\end{array}$$

□

Lemma B.5. *Let e be an expression, ϱ an expression substitution and $[\sigma_j]_{j\in J}$ a set of type substitutions such that $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$. Then $(e\varrho)@[\sigma_j]_{j\in J} = (e@[\sigma_j]_{j\in J})\varrho$.*

Proof. By induction on the structure of e.

$\underline{e = c}$**:**

$$\begin{array}{rcl}(c\varrho)@[\sigma_j]_{j\in J} &=& c@[\sigma_j]_{j\in J}\\ &=& c\\ &=& c\varrho\\ &=& (c@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = x}$**:** if $x \notin \mathsf{dom}(\varrho)$, then

$$\begin{array}{rcl}(x\varrho)@[\sigma_j]_{j\in J} &=& x@[\sigma_j]_{j\in J}\\ &=& x\\ &=& x\varrho\\ &=& (x@[\sigma_j]_{j\in J})\varrho\end{array}$$

Otherwise, let $\varrho(x) = e'$. As $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$, we have $\mathsf{tv}(e') \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$. Then

$$\begin{array}{rcl}(x\varrho)@[\sigma_j]_{j\in J} &=& e'@[\sigma_j]_{j\in J}\\ &=& e'\\ &=& x\varrho\\ &=& (x@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = (e_1, e_2)}$**:**

$$\begin{array}{rcll}((e_1,e_2)\varrho)@[\sigma_j]_{j\in J} &=& (e_1\varrho, e_2\varrho)@[\sigma_j]_{j\in J}\\ &=& ((e_1\varrho)@[\sigma_j]_{j\in J}, (e_2\varrho)@[\sigma_j]_{j\in J})\\ &=& ((e_1@[\sigma_j]_{j\in J})\varrho, (e_2@[\sigma_j]_{j\in J})\varrho) & \text{(by induction)}\\ &=& (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J})\varrho\\ &=& ((e_1,e_2)@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = \pi_i(e')}$**:**

$$\begin{array}{rcll}(\pi_i(e')\varrho)@[\sigma_j]_{j\in J} &=& (\pi_i(e'\varrho))@[\sigma_j]_{j\in J}\\ &=& \pi_i((e'\varrho)@[\sigma_j]_{j\in J})\\ &=& \pi_i((e_1@[\sigma_j]_{j\in J})\varrho) & \text{(by induction)}\\ &=& \pi_i(e'@[\sigma_j]_{j\in J})\varrho\\ &=& ((\pi_i(e'))@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = e_1e_2}$**:**

$$\begin{array}{rcll}((e_1e_2)\varrho)@[\sigma_j]_{j\in J} &=& ((e_1\varrho)(e_2\varrho))@[\sigma_j]_{j\in J}\\ &=& ((e_1\varrho)@[\sigma_j]_{j\in J})((e_2\varrho)@[\sigma_j]_{j\in J})\\ &=& ((e_1@[\sigma_j]_{j\in J})\varrho)((e_2@[\sigma_j]_{j\in J})\varrho) & \text{(by induction)}\\ &=& ((e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}))\varrho\\ &=& ((e_1e_2)@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e'}$**:**

$$\begin{array}{rcll}((\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e')\varrho)@[\sigma_j]_{j\in J} &=& (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.(e'\varrho))@[\sigma_j]_{j\in J}\\ &=& \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.(e'\varrho)\\ &=& (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e')\varrho & \text{(because } \mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset\text{)}\\ &=& ((\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e')@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = e_0 \in t\,?\,e_1 : e_2}$**:**

$$\begin{array}{rcll}&& ((e_0\in t\,?\,e_1 : e_2)\varrho)@[\sigma_j]_{j\in J}\\ &=& ((e_0\varrho)\in t\,?\,(e_1\varrho) : (e_2\varrho))@[\sigma_j]_{j\in J}\\ &=& ((e_0\varrho)@[\sigma_j]_{j\in J})\in t\,?\,((e_1\varrho)@[\sigma_j]_{j\in J}) : ((e_2\varrho)@[\sigma_j]_{j\in J})\\ &=& ((e_0@[\sigma_j]_{j\in J})\varrho)\in t\,?\,((e_1@[\sigma_j]_{j\in J})\varrho) : ((e_2@[\sigma_j]_{j\in J})\varrho) & \text{(by induction)}\\ &=& ((e_0@[\sigma_j]_{j\in J})\in t\,?\,(e_1@[\sigma_j]_{j\in J}) : (e_2@[\sigma_j]_{j\in J}))\varrho\\ &=& ((e_0\in t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J})\varrho\end{array}$$

$\underline{e = e'[\sigma_k]_{k\in K}}$**:** using α-conversion on the polymorphic type variables of e, we can assume $\mathsf{tv}(\varrho) \cap \bigcup_{k\in K} \mathsf{dom}(\sigma_k) = \emptyset$. Consequently we have $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J,k\in K} \mathsf{dom}(\sigma_j \circ \sigma_k) = \emptyset$, and we deduce

$$\begin{array}{rcll}((e'[\sigma_k]_{k\in K})\varrho)@[\sigma_j]_{j\in J} &=& ((e'\varrho)[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J}\\ &=& (e'\varrho)@([\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K})\\ &=& (e'\varrho)@[\sigma_j\circ\sigma_k]_{j\in J,k\in K}\\ &=& (e'@[\sigma_j\circ\sigma_k]_{j\in J,k\in K})\varrho & \text{(by induction)}\\ &=& (e'@([\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}))\varrho\\ &=& ((e'[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J})\varrho\end{array}$$

□

Lemma B.6 ([Expression] substitution lemma). *Let $e, e_1, \ldots, e_n$ be expressions, $x_1, \ldots, x_n$ distinct variables, and $t, t_1, \ldots, t_n$ types. If $\Delta \fatsemi \Gamma, (x_1 : t_1), \ldots, (x_n : t_n) \vdash e : t$ and $\Delta \fatsemi \Gamma \vdash e_i : t_i$ for all i, then $\Delta \fatsemi \Gamma \vdash e\{e_1/x_1, \ldots, e_n/x_n\} : t$.*

Proof. By induction on the typing derivations for $\Delta \fatsemi \Gamma, (x_1 : t_1), \ldots, (x_n : t_n) \vdash e : t$. We simply "plug" a copy of the derivation for $\Delta \fatsemi \Gamma \vdash e_i : t_i$ wherever the rule (*var*) is used for variable x_i. For simplicity, in what follows, we write Γ' for $\Gamma, (x_1 : t_1), \ldots, (x_n : t_n)$ and ϱ for $\{e_1/x_1, \ldots, e_n/x_n\}$. We proceed by a case analysis on the last applied rule.

(*const*)**:** straightforward.

(*var*)**:** $e = x$ and $\Delta \wr \Gamma' \vdash x : \Gamma'(x)$.

If $x = x_i$, then $\Gamma'(x) = t_i$ and $x\varrho = e_i$. From the premise, we have $\Delta \wr \Gamma \vdash e_i : t_i$. The result follows. Otherwise, $\Gamma'(x) = \Gamma(x)$ and $x\varrho = x$. Clearly, we have $\Delta \wr \Gamma \vdash x : \Gamma(x)$. Thus the result follows as well.

(*pair*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma' \vdash e_1 : t_1} \quad \dfrac{\cdots}{\Delta \wr \Gamma' \vdash e_1 : t_1}}{\Delta \wr \Gamma' \vdash (e_1, e_2) : (t_1 \times t_2)}\ (pair)$$

By applying the induction hypothesis twice, we have $\Delta \wr \Gamma \vdash e_i\varrho : t_i$. By (*pair*), we get $\Delta \wr \Gamma \vdash (e_1\varrho, e_2\varrho) : (t_1 \times t_2)$, that is, $\Delta \wr \Gamma \vdash (e_1, e_2)\varrho : (t_1 \times t_2)$.

(*proj*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma' \vdash e' : t_1 \times t_2}}{\Delta \wr \Gamma' \vdash \pi_i(e') : t_i}\ (proj)$$

By induction, we have $\Delta \wr \Gamma \vdash e'\varrho : t_1 \times t_2$. Then the rule (*proj*) gives us $\Delta \wr \Gamma \vdash \pi_i(e'\varrho) : t_i$, that is $\Delta \wr \Gamma \vdash \pi_i(e')\varrho : t_i$.

(*abstr*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\begin{array}{l}\forall i \in I, j \in J.\ \Delta' \wr \Gamma', (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j)\end{array}}}{\Delta \wr \Gamma' \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ (abstr)$$

By α-conversion, we can ensure that $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$. By induction, we have $\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash (e'@[\sigma_j])\varrho : s_i\sigma_j$ for all $i \in I$ and $j \in J$. Because $\mathsf{tv}(\varrho) \cap \mathsf{dom}(\sigma_j) = \emptyset$, by Lemma B.5, we get $\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash (e'\varrho)@[\sigma_j] : s_i\sigma_j$. Then by applying (*abstr*), we obtain $\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.(e'\varrho) : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$. That is, $\Delta \wr \Gamma \vdash (\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e)\varrho : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$ (because $\mathsf{tv}(\varrho) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$).

(*case*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma' \vdash e_0 : t'} \quad \begin{cases} t' \not\leq \neg t \ \Rightarrow \ \dfrac{\cdots}{\Delta \wr \Gamma' \vdash e_1 : s} \\ t' \not\leq t \ \Rightarrow \ \dfrac{\cdots}{\Delta \wr \Gamma' \vdash e_2 : s} \end{cases}}{\Delta \wr \Gamma' \vdash (e_0 \in t\ ?\ e_1 : e_2) : s}\ (case)$$

By induction, we have $\Delta \wr \Gamma \vdash e_0\varrho : t'$ and $\Delta \wr \Gamma \vdash e_i\varrho : s$ (for i such that $\Delta \wr \Gamma' \vdash e_i : s$ has been type-checked in the original derivation). Then the rule (*case*) gives us $\Delta \wr \Gamma \vdash (e_0\varrho \in t\ ?\ e_1\varrho : e_2\varrho) : s$ that is $\Delta \wr \Gamma \vdash (e_0 \in t\ ?\ e_1 : e_2)\varrho : s$.

(*inst*)**:**

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma' \vdash e' : s} \quad \sigma \sharp \Delta}{\Delta \wr \Gamma' \vdash e'[\sigma] : s\sigma}\ (inst)$$

Using α-conversion on the polymorphic type variables of e, we can assume $\mathsf{tv}(\varrho) \cap \mathsf{dom}(\sigma) = \emptyset$. By induction, we have $\Delta \wr \Gamma \vdash e'\varrho : s$. Since $\sigma \sharp \Delta$, by applying (*inst*) we obtain $\Delta \wr \Gamma \vdash (e'\varrho)[\sigma] : s\sigma$, that is, $\Delta \wr \Gamma \vdash (e'[\sigma])\varrho : s\sigma$ because $\mathsf{tv}(\varrho) \cap \mathsf{dom}(\sigma) = \emptyset$.

(*inter*)**:**

$$\frac{\forall j \in J.\ \dfrac{\cdots}{\Delta \wr \Gamma' \vdash e'[\sigma_j] : t_j}}{\Delta \wr \Gamma' \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (inter)$$

By induction, for all $j \in J$ we have $\Delta \wr \Gamma \vdash (e'[\sigma_j])\varrho : t_j$, that is $\Delta \wr \Gamma \vdash (e'\varrho)[\sigma_j] : t_j$. Then by applying (*inter*) we get $\Delta \wr \Gamma \vdash (e'\varrho)[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j$, that is $\Delta \wr \Gamma \vdash (e'[\sigma_j]_{j\in J})\varrho : \bigwedge_{j\in J} t_j$.

(*subsum*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma' \vdash e' : s} \quad s \leq t}{\Delta \wr \Gamma' \vdash e' : t}\ (subsum)$$

By induction, we have $\Delta \wr \Gamma \vdash e'\varrho : s$. Then the rule (*subsum*) gives us $\Delta \wr \Gamma \vdash e'\varrho : t$.

□

Definition B.7. *Given two typing environments Γ_1, Γ_2, we define their* intersection *as*

$$(\Gamma_1 \wedge \Gamma_2)(x) = \begin{cases} \Gamma_1(x) \wedge \Gamma_2(x) & \text{if } x \in \mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

We define $\Gamma_2 \leq \Gamma_1$ if $\Gamma_2(x) \leq \Gamma_1(x)$ for all $x \in \mathsf{dom}(\Gamma_1)$, and $\Gamma_1 \simeq \Gamma_2$ if $\Gamma_1 \leq \Gamma_2$ and $\Gamma_2 \leq \Gamma_1$.

Given an expression e and a set Δ of (monomorphic) type variables, we write $e \sharp \Delta$ if $\sigma_j \sharp \Delta$ for all the type substitution σ_j that occur in a subterm of e of the form $e'[\sigma_j]_{j\in J}$ (in other terms, we do not consider the substitutions that occur in the decorations of λ-abstractions).

Lemma B.8 (Weakening). *Let e be an expression, Γ, Γ' two typing environments and Δ' a set of type variables. If $\Delta \fatsemi \Gamma \vdash e : t$, $\Gamma' \leq \Gamma$ and $e \sharp \Delta'$, then $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e : t$.*

Proof. By induction on the derivation of $\Delta \fatsemi \Gamma \vdash e : t$. We perform a case analysis on the last applied rule.

(*const*)**:** straightforward.

(*var*)**:** $\Delta \fatsemi \Gamma \vdash x : \Gamma(x)$. It is clear that $\Delta \cup \Delta' \fatsemi \Gamma' \vdash x : \Gamma'(x)$ by (*var*). Since $\Gamma'(x) \leq \Gamma(x)$, by (*subsum*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash x : \Gamma(x)$.

(*pair*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

By applying the induction hypothesis twice, we have $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e_i : t_i$. Then by (*pair*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash (e_1, e_2) : t_1 \times t_2$.

(*proj*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : t_1 \times t_2}}{\Delta \fatsemi \Gamma \vdash \pi_i(e') : t_i}\ (proj)$$

By the induction hypothesis, we have $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e' : t_1 \times t_2$. Then by (*proj*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash \pi_i(e') : t_i$.

(*abstr*)**:** consider the following derivation:

$$\frac{\begin{array}{l}\forall i \in I, j \in J.\ \dfrac{\cdots}{\Delta'' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j} \\ \Delta'' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j)\end{array}}{\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j}\ (abstr)$$

By induction, we have $\Delta'' \cup \Delta' \fatsemi \Gamma', (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j$ for all $i \in I$ and $j \in J$. Then by (*abstr*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$.

(*case*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_0 : t'} \quad \begin{cases} t' \not\leq \neg t \quad \Rightarrow \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : s} \\ t' \not\leq t \quad \Rightarrow \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e_0 \in t\,?\,e_1 : e_2) : s}\ (case)$$

By induction, we have $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e_0 : t_0$ and $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e_i : s$ (for i such that e_i has been type-checked in the original derivation). Then by (*case*), we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash (e_0 \in t\,?\,e_1 : e_2) : s$.

(*inst*)**:** consider the following derivation:

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : s} \quad \sigma \sharp \Delta}{\Delta \fatsemi \Gamma \vdash e'[\sigma] : s\sigma}\ (inst)$$

By induction, we have $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e' : s$. Since $e \sharp \Delta'$(*i.e.*, $e'[\sigma] \sharp \Delta'$), we have $\sigma \sharp \Delta'$. Then $\sigma \sharp \Delta \cup \Delta'$. Therefore, by applying (*inst*) we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e'[\sigma] : s\sigma$.

(*inter*)**:** consider the following derivation:

$$\frac{\forall j \in J.\ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e'[\sigma_j] : t_j}}{\Delta \fatsemi \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (inter)$$

By induction, we have $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e'[\sigma_j] : t_j$ for all $j \in J$. Then the rule (*inst*) gives us $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j$.

(*subsum*)**:** there exists a type s such that

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : s} \quad s \leq t}{\Delta \fatsemi \Gamma \vdash e' : t}\ (subsum)$$

By induction, we have $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e' : s$. Then by applying the rule (*subsum*) we get $\Delta \cup \Delta' \fatsemi \Gamma' \vdash e' : t$.

□

The next two lemmas are used to simplify sets of type-substitutions applied to expressions when they are redundant or they work on variables that are not in the expressions.

Lemma B.9 (Useless Substitutions). *Let e be an expression and $[\sigma_k]_{k\in K}$, $[\sigma'_k]_{k\in K}$ two sets of substitutions such that $\mathsf{dom}(\sigma'_k) \cap \mathsf{dom}(\sigma_k) = \emptyset$ and $\mathsf{dom}(\sigma'_k) \cap \mathsf{tv}(e) = \emptyset$ for all $k \in K$. Then*

$$\Delta \fatsemi \Gamma \vdash e@[\sigma_k]_{k\in K} : t \iff \Delta \fatsemi \Gamma \vdash e@[\sigma_k \cup \sigma'_k]_{k\in K} : t$$

Proof. Straightforward. □

Henceforth we use "$\uplus$" to denote the union of multi-sets (*e.g.*, $\{1,2\} \uplus \{1,3\} = \{1,2,1,3\}$).

Lemma B.10 (Redundant Substitutions). *Let $[\sigma_j]_{j\in J}$ and $[\sigma_j]_{j\in J'}$ be two sets of substitutions such that $J' \subseteq J$. Then*

$$\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J\uplus J'} : t \iff \Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J} : t$$

Proof. Similar to Lemma B.9. □

Lemma B.9 states that if a type variable α in the domain of a type substitution σ does not occur in the applied expression e, namely, $\alpha \in \mathsf{dom}(\sigma) \setminus \mathsf{tv}(e)$, then that part of the substitution is useless and can be safely eliminated. Lemma B.10 states that although our $[\sigma_j]_{j\in J}$ are formally multisets of type-substitutions, in practice they behave as sets, since repeated entries of type substitutions can be safely removed. Therefore, to simplify an expression without altering its type (and semantics), we first eliminate the useless type variables, yielding concise type substitutions, and then remove the redundant type substitutions. It explains why we do not apply relabeling when the domains of the type substitutions do not contain type variables in expressions in Definition A.11.

Moreover, Lemma B.10 also indicates that it is safe to keep only the type substitutions which are different from each other when we merge two sets of substitutions (*e.g.* Lemmas B.13 and B.14). In what follows, without explicit mention, we assume that there are no useless type variables in the domain of any type substitution and no redundant type substitutions in any set of type substitutions.

Lemma B.11 (Relabeling). *Let e be an expression, $[\sigma_j]_{j\in J}$ a set of type substitutions and Δ a set of type variables such that $\sigma_j \sharp \Delta$ for all $j \in J$. If $\Delta \fatsemi \Gamma \vdash e : t$, then*

$$\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j$$

Proof. The proof proceeds by induction and case analysis on the structure of e. For each case we use an auxiliary internal induction on the typing derivation. We label **E** the main (external) induction and **I** the internal induction in what follows.

$\underline{e = c}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*const*) or (*subsum*). Assume that the typing derivation ends with (*const*). Trivially, we have $\Delta \fatsemi \Gamma \vdash c : b_c$. Since $c@[\sigma_j]_{j\in J} = c$ and $b_c \simeq \bigwedge_{j\in J} b_c\sigma_j$, by subsumption, we have $\Delta \fatsemi \Gamma \vdash c@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} b_c\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*):

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : s} \qquad s \leq t}{\Delta \fatsemi \Gamma \vdash e : t}\ (subsum)$$

Then by **I**-induction, we have $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Since $s \leq t$, we get $\bigwedge_{j\in J} s\sigma_j \leq \bigwedge_{j\in J} t\sigma_j$. Then by applying the rule (*subsum*), we have $\Delta \fatsemi \Gamma \vdash e@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j$.

$\underline{e = x}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*var*) or (*subsum*). Assume that the typing derivation ends with (*var*). Trivially, by (*var*), we get $\Delta \fatsemi \Gamma \vdash x : \Gamma(x)$. Moreover, we have $x@[\sigma_j]_{j\in J} = x$ and $\Gamma(x) = \bigwedge_{j\in J} \Gamma(x)\sigma_j$ (as $\mathsf{var}(\Gamma) \subseteq \Delta$). Therefore, we deduce that $\Delta \fatsemi \Gamma \vdash x@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} \Gamma(x)\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = (e_1, e_2)}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*pair*) or (*subsum*). Assume that the typing derivation ends with (*pair*):

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1} \qquad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash e_i@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_i\sigma_j$. Then by (*pair*), we get $\Delta \fatsemi \Gamma \vdash (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J}) : (\bigwedge_{j\in J} t_1\sigma_j \times \bigwedge_{j\in J} t_2\sigma_j)$, that is, $\Delta \fatsemi \Gamma \vdash (e_1, e_2)@[\sigma_j]_{j\in J} :$

$\bigwedge_{j\in J}(t_1 \times t_2)\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \pi_i(e')}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*proj*) or (*subsum*). Assume that the typing derivation ends with (*proj*):

$$\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : t_1 \times t_2}}{\Delta \fatsemi \Gamma \vdash \pi_i(e') : t_i}\ (proj)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash e'@[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(t_1 \times t_2)\sigma_j$, that is, $\Delta \fatsemi \Gamma \vdash e'@[\sigma_j]_{j\in J} : (\bigwedge_{j\in J} t_1\sigma_j \times \bigwedge_{j\in J} t_2\sigma_j)$. Then the rule (*proj*) gives us that $\Delta \fatsemi \Gamma \vdash \pi_i(e'@[\sigma_j]_{j\in J}) : \bigwedge_{j\in J} t_i\sigma_j$, that is, $\Delta \fatsemi \Gamma \vdash \pi_i(e')@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_i\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e_1e_2}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*appl*) or (*subsum*). Assume that the typing derivation ends with (*appl*):

$$\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t \to s} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t}}{\Delta \fatsemi \Gamma \vdash e_1e_2 : s}\ (pair)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash e_1@[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(t \to s)\sigma_j$ and $\Delta \fatsemi \Gamma \vdash e_2@[\sigma_j] : \bigwedge_{j\in J} t\sigma_j$. Since $\bigwedge_{j\in J}(t \to s)\sigma_j \leq (\bigwedge_{j\in J} t\sigma_j) \to (\bigwedge_{j\in J} s\sigma_j)$, by (*subsum*), we have $\Delta \fatsemi \Gamma \vdash e_1@[\sigma_j]_{j\in J} : (\bigwedge_{j\in J} t\sigma_j) \to (\bigwedge_{j\in J} s\sigma_j)$. Then by (*appl*), we get

$$\Delta \fatsemi \Gamma \vdash (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J}) : \bigwedge_{j\in J} s\sigma_j$$

that is, $\Delta \fatsemi \Gamma \vdash (e_1e_2)@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e'}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*abstr*) or (*subsum*). Assume that the typing derivation ends with (*abstr*):

$$\dfrac{\begin{array}{l}\forall i \in I, k \in K.\ \dfrac{\cdots}{\Delta' \fatsemi \Gamma, (x : t_i\sigma_k) \vdash e'@[\sigma_k] : s_i\sigma_k} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I,k\in K} t_i\sigma_k \to s_i\sigma_k)\end{array}}{\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e' : \bigwedge_{i\in I,k\in K} t_i\sigma_k \to s_i\sigma_k}\ (abstr)$$

Using α-conversion, we can assume that $\sigma_j \sharp (\mathsf{var}(\bigwedge_{i\in I,k\in K} t_i\sigma_k \to s_i\sigma_k) \setminus \Delta)$ for $j \in J$. Hence $\sigma_j \sharp \Delta'$. By **E**-induction, we have

$$\Delta' \fatsemi \Gamma, (x : (t_i\sigma_k)) \vdash (e'@[\sigma_k])@[\sigma_j] : (s_i\sigma_k)\sigma_j$$

for all $i \in I$, $k \in K$ and $j \in J$. By Lemma B.4, $(e'@[\sigma_k])@[\sigma_j] = e'@([\sigma_j] \circ [\sigma_k])$. So

$$\Delta' \fatsemi \Gamma, (x : (t_i\sigma_k)) \vdash e'@([\sigma_j] \circ [\sigma_k]) : (s_i\sigma_k)\sigma_j$$

Finally, by (*abstr*), we get

$$\Delta \fatsemi \Gamma \vdash \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j\circ\sigma_k]_{j\in J,k\in K}} x.e' : \bigwedge_{i\in I,j\in J,k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)$$

that is,

$$\Delta \fatsemi \Gamma \vdash (\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e')@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} (\bigwedge_{i\in I,k\in K} t_i\sigma_k \to s_i\sigma_k)\sigma_j$$

Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e' \in t\,?\,e_1 : e_2}$**:** the typing derivation $\Delta \fatsemi \Gamma \vdash e : t$ should end with either (*case*) or (*subsum*). Assume that the typing derivation ends with (*case*):

$$\dfrac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e' : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \ \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e' \in t\,?\,e_1 : e_2) : s}\ (case)$$

By **E**-induction, we have $\Delta \fatsemi \Gamma \vdash e'@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j$. Suppose $\bigwedge_{j\in J} t'\sigma_j \not\leq \neg t$; then we must have $t' \not\leq \neg t$, and the branch for e_1 has been type-checked. By the **E**-induction hypothesis, we have $\Delta \fatsemi \Gamma \vdash e_1@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Similarly, if $\bigwedge_{j\in J} t'\sigma_j \not\leq t$, then the second branch e_2 has been type-checked, and we have $\Delta \fatsemi \Gamma \vdash e_2@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$ by the **E**-induction hypothesis. By (*case*),

we have

$$\Delta \wr \Gamma \vdash (e'@[\sigma_j]_{j\in J} \in t \,?\, e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J}) : \bigwedge_{j\in J} s\sigma_j$$

that is $\Delta \wr \Gamma \vdash (e' \in t \,?\, e_1 : e_2)@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma]}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*inst*) or (*subsum*). Assume that the typing derivation ends with (*inst*):

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : t} \quad \sigma \sharp \Delta}{\Delta \wr \Gamma \vdash e'[\sigma] : t\sigma}\ (inst)$$

Consider the set of substitutions $[\sigma_j \circ \sigma]_{j\in J}$. It is clear that $\sigma_j \circ \sigma \sharp \Delta$ for all $j \in J$. By **E**-induction, we have

$$\Delta \wr \Gamma \vdash e'@[\sigma_j \circ \sigma]_{j\in J} : \bigwedge_{j\in J} t(\sigma_j \circ \sigma)$$

that is, $\Delta \wr \Gamma \vdash (e'[\sigma])@[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(t\sigma)\sigma_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma_k]_{k\in K}}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*inter*) or (*subsum*). Assume that the typing derivation ends with (*inter*):

$$\dfrac{\forall k \in K.\ \dfrac{\cdots}{\Delta \wr \Gamma \vdash e'[\sigma_k] : t_k}}{\Delta \wr \Gamma \vdash e'[\sigma_k]_{k\in K} : \bigwedge_{k\in K} t_k}\ (inter)$$

As an intermediary result, we first prove that the derivation can be rewritten as

$$\dfrac{\dfrac{\forall k \in K.\ \dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : s} \quad \sigma_k \sharp \Delta}{\Delta \wr \Gamma \vdash e'[\sigma_k] : s\sigma_k}\ (inst)}{\Delta \wr \Gamma \vdash e'[\sigma_k]_{k\in K} : \bigwedge_{k\in K} s\sigma_k}\ (inter) \qquad \bigwedge_{k\in K} s\sigma_k \leq \bigwedge_{k\in K} t_k}{\Delta \wr \Gamma \vdash e'[\sigma_k]_{k\in K} : \bigwedge_{k\in K} t_k}\ (subsum)$$

We proceed by induction on the original derivation. It is clear that each sub-derivation $\Delta \wr \Gamma \vdash e'[\sigma_k] : t_k$ ends with either (*inst*) or (*subsum*). If all the sub-derivations end with an instance of (*inst*), then for all $k \in K$, we have

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : s_k} \quad \sigma_k \sharp \Delta}{\Delta \wr \Gamma \vdash e'[\sigma_k] : s_k\sigma_k}\ (inst)$$

By Lemma B.2, we have $\Delta \wr \Gamma \vdash e' : \bigwedge_{k\in K} s_k$. Let $s = \bigwedge_{k\in K} s_k$. Then by (*inst*), we get $\Delta \wr \Gamma \vdash e'[\sigma_k] : s\sigma_k$. Finally, by (*inter*) and (*subsum*), the intermediary result holds. Otherwise, at least one of the sub-derivations ends with an instance of (*subsum*); the intermediary result also holds by induction.
Now that the intermediary result is proved, we go back to the proof of the lemma. Consider the set of substitutions $[\sigma_j \circ \sigma_k]_{j\in J,k\in K}$. It is clear that $\sigma_j \circ \sigma_k \sharp \Delta$ for all $j \in J, k \in K$. By **E**-induction on e' (*i.e.*, $\Delta \wr \Gamma \vdash e' : s$), we have

$$\Delta \wr \Gamma \vdash e'@[\sigma_j \circ \sigma_k]_{j\in J,k\in K} : \bigwedge_{j\in J,k\in K} s(\sigma_j \circ \sigma_k)$$

that is, $\Delta \wr \Gamma \vdash (e'[\sigma_k]_{k\in K})@[\sigma_j]_{j\in J} : \bigwedge_{j\in J}(\bigwedge_{k\in K} s\sigma_k)\sigma_j$. As $\bigwedge_{k\in K} s\sigma_k \leq \bigwedge_{k\in K} t_k$, we get $\bigwedge_{j\in J}(\bigwedge_{k\in K} s\sigma_k)\sigma_j \leq \bigwedge_{j\in J}(\bigwedge_{k\in K} t_k)\sigma_j$. Then by (*subsum*), the result follows.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

□

Corollary B.12. *If* $\Delta \wr \Gamma \vdash e[\sigma_j]_{j\in J} : t$*, then* $\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : t$*.*

Proof. Immediate consequence of Lemma B.11. □

Lemma B.13. *If* $\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : t$ *and* $\Delta' \wr \Gamma' \vdash e@[\sigma_j]_{j\in J'} : t'$*, then* $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j\in J\cup J'} : t \wedge t'$

Proof. The proof proceeds by induction and case analysis on the structure of e. For each case we use an auxiliary internal induction on both typing derivations. We label **E** the main (external) induction and **I** the internal induction in what follows.

$\underline{e = c}$**:** $e@[\sigma_j]_{j\in J} = e@[\sigma_j]_{j\in J'} = c$. Clearly, both typing derivations should end with either (*const*) or (*subsum*). Assume that both derivations end with (*const*):

$$\frac{}{\Delta \wr \Gamma \vdash c : b_c}\ (const) \qquad \frac{}{\Delta' \wr \Gamma' \vdash c : b_c}\ (const)$$

Trivially, by (*const*) we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash c : b_c$, that is $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j\in J\cup J'} : b_c$. As $b_c \simeq b_c \wedge b_c$, by (*subsum*), the result follows.

Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), for instance,

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : s} \qquad s \leq t}{\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : t}\ (subsum)$$

Then by **I**-induction on $\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J} : s$ and $\Delta' \wr \Gamma' \vdash e@[\sigma_j]_{j\in J'} : t'$, we have

$$\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j\in J\cup J'} : s \wedge t'$$

Since $s \leq t$, we have $s \wedge t' \leq t \wedge t'$. By (*subsum*), the result follows as well.

$\underline{e = x}$**:** $e@[\sigma_j]_{j\in J} = e@[\sigma_j]_{j\in J'} = x$. Clearly, both typing derivations should end with either (*var*) or (*subsum*). Assume that both derivations end with an instance of (*var*):

$$\frac{}{\Delta \wr \Gamma \vdash x : \Gamma(x)}\ (var) \qquad \frac{}{\Delta' \wr \Gamma' \vdash x : \Gamma'(x)}\ (var)$$

Since $x \in \mathsf{dom}(\Gamma)$ and $x \in \mathsf{dom}(\Gamma')$, $x \in \mathsf{dom}(\Gamma \wedge \Gamma')$. By (*var*), we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash x : (\Gamma \wedge \Gamma')(x)$, that is, $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j\in J\cup J'} : \Gamma(x) \wedge \Gamma'(x)$.

Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = (e_1, e_2)}$**:** $e@[\sigma_j]_{j\in J} = (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J})$ and $e@[\sigma_j]_{j\in J'} = (e_1@[\sigma_j]_{j\in J'}, e_2@[\sigma_j]_{j\in J'})$. Clearly, both typing derivations should end with either (*pair*) or (*subsum*). Assume that both derivations end with an instance of (*pair*):

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1@[\sigma_j]_{j\in J} : s_1} \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_2@[\sigma_j]_{j\in J} : s_2}}{\Delta \wr \Gamma \vdash (e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J}) : (s_1 \times s_2)}\ (pair)$$

$$\frac{\dfrac{\cdots}{\Delta' \wr \Gamma' \vdash e_1@[\sigma_j]_{j\in J'} : s'_1} \quad \dfrac{\cdots}{\Delta' \wr \Gamma' \vdash e_2@[\sigma_j]_{j\in J'} : s'_2}}{\Delta' \wr \Gamma' \vdash (e_1@[\sigma_j]_{j\in J'}, e_2@[\sigma_j]_{j\in J'}) : (s'_1 \times s'_2)}\ (pair)$$

By **E**-induction, we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_i@[\sigma_j]_{j\in J\cup J'} : s_i \wedge s'_i$. Then by (*pair*), we get $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_1@[\sigma_j]_{j\in J\cup J'}, e_2@[\sigma_j]_{j\in J\cup J'}) : (s_1 \wedge s'_1) \times (s_2 \wedge s'_2)$, that is $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_1, e_2)@[\sigma_j]_{j\in J\cup J'} : (s_1 \wedge s'_1) \times (s_2 \wedge s'_2)$. Moreover, because intersection distributes over product, we have $(s_1 \wedge s'_1) \times (s_2 \wedge s'_2) \simeq (s_1 \times s_2) \wedge (s'_1 \times s'_2)$. Finally, by applying (*subsum*), we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_1, e_2)@[\sigma_j]_{j\in J\cup J'} : (s_1 \times s_2) \wedge (s'_1 \times s'_2)$.

Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \pi_i(e')}$**:** $e@[\sigma_j]_{j\in J} = \pi_i(e'@[\sigma_j]_{j\in J})$ and $e@[\sigma_j]_{j\in J'} = \pi_i(e'@[\sigma_j]_{j\in J'})$, where $i = 1, 2$. Clearly, both typing derivations should end with either (*proj*) or (*subsum*). Assume that both derivations end with an instance of (*proj*):

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e'@[\sigma_j]_{j\in J} : s_1 \times s_2}}{\Delta \wr \Gamma \vdash \pi_i(e'@[\sigma_j]_{j\in J}) : s_i}\ (proj) \qquad \frac{\dfrac{\cdots}{\Delta' \wr \Gamma' \vdash e'@[\sigma_j]_{j\in J'} : s'_1 \times s'_2}}{\Delta' \wr \Gamma' \vdash \pi_i(e'@[\sigma_j]_{j\in J'}) : s'_i}\ (proj)$$

By **E**-induction, we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e'@[\sigma_j]_{j\in J\cup J'} : (s_1 \times s_2) \wedge (s'_1 \times s'_2)$. Since $(s_1 \times s_2) \wedge (s'_1 \times s'_2) \simeq (s_1 \wedge s'_1) \times (s_2 \wedge s'_2)$ (See the case of $e = (e_1, e_2)$), by (*subsum*), we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e'@[\sigma_j]_{j\in J\cup J'} : (s_1 \wedge s'_1) \times (s_2 \wedge s'_2)$. Finally, by applying (*proj*), we get $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash \pi_i(e@[\sigma_j]_{j\in J\cup J'}) : s_i \wedge s'_i$, that is $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash \pi_i(e)@[\sigma_j]_{j\in J\cup J'} : s_i \wedge s'_i$.

Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e_1 e_2}$**:** $e@[\sigma_j]_{j\in J} = (e_1@[\sigma_j]_{j\in J})(e_2@[\sigma_j]_{j\in J})$ and $e@[\sigma_j]_{j\in J'} = (e_1@[\sigma_j]_{j\in J'})(e_2@[\sigma_j]_{j\in J'})$. Clearly, both typing derivations should end with either (*appl*) or (*subsum*). Assume that both derivations end with an instance of (*appl*):

$$\frac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1@[\sigma_j]_{j\in J} : s_1 \to s_2} \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_2@[\sigma_j]_{j\in J} : s_1}}{\Delta \wr \Gamma \vdash (e_1@[\sigma_j]_{j\in J})\ (e_2@[\sigma_j]_{j\in J}) : s_2}\ (appl)$$

$$\frac{\dfrac{\cdots}{\Delta' \wr \Gamma' \vdash e_1@[\sigma_j]_{j\in J'} : s'_1 \to s'_2} \quad \dfrac{\cdots}{\Delta' \wr \Gamma' \vdash e_2@[\sigma_j]_{j\in J'} : s'_1}}{\Delta' \wr \Gamma' \vdash (e_1@[\sigma_j]_{j\in J'})\ (e_2@[\sigma_j]_{j\in J'}) : s'_2}\ (appl)$$

By **E**-induction, we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_1@[\sigma_j]_{j\in J\cup J'} : (s_1 \to s_2) \wedge (s'_1 \to s'_2)$ and $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_2@[\sigma_j]_{j\in J\cup J'} : s_1 \wedge s'_1$. Because intersection distributes over arrows, we

have $(s_1 \to s_2) \wedge (s_1' \to s_2') \leq (s_1 \wedge s_1') \to (s_2 \wedge s_2')$. Then by the rule (*subsum*), we get $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_1@[\sigma_j]_{j\in J\cup J'} : (s_1 \wedge s_1') \to (s_2 \wedge s_2')$. Finally by (*appl*), we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_1@[\sigma_j]_{j\in J\cup J'})(e_2@[\sigma_j]_{j\in J\cup J'}) : s_2 \wedge s_2'$, that is, $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_1e_2)@[\sigma_j]_{j\in J\cup J'} : s_2 \wedge s_2'$.
Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e'}$**:** $e@[\sigma_j]_{j\in J} = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e'$ and $e@[\sigma_j]_{j\in J'} = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J'}\circ[\sigma_k]_{k\in K}} x.e'$. Clearly, both typing derivations should end with either (*abstr*) or (*subsum*). Assume that both derivations end with an instance of (*abstr*):

$$\dfrac{\begin{array}{l}\forall i \in I, j \in J, k \in K.\ \dfrac{\ldots}{\Delta_1 \wr \Gamma, (x : t_i(\sigma_j \circ \sigma_k)) \vdash e'@[\sigma_j \circ \sigma_k] : s_i(\sigma_j \circ \sigma_k)} \\ \Delta_1 = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J, k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k))\end{array}}{\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}\circ[\sigma_k]_{k\in K}} x.e' : \bigwedge_{i\in I, j\in J, k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)}$$

$$\dfrac{\begin{array}{l}\forall i \in I, j \in J', k \in K.\ \dfrac{\ldots}{\Delta_2 \wr \Gamma', (x : t_i(\sigma_j \circ \sigma_k)) \vdash e'@[\sigma_j \circ \sigma_k] : s_i(\sigma_j \circ \sigma_k)} \\ \Delta_2 = \Delta' \cup \mathsf{var}(\bigwedge_{i\in I, j\in J', k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k))\end{array}}{\Delta' \wr \Gamma' \vdash \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J'}\circ[\sigma_k]_{k\in K}} x.e' : \bigwedge_{i\in I, j\in J', k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)}$$

Consider any expression $e'@([\sigma_j]\circ[\sigma_k])$ and any $e_0[\sigma_{j_0}]_{j_0\in J_0}$ in $e'@([\sigma_j]\circ[\sigma_k])$, where $j \in J\cup J', k \in K$. (Then $e_0[\sigma_{j_0}]_{j_0\in J_0}$ must be from e'). All type variables in $\bigcup_{j_0\in J_0} \mathsf{dom}(\sigma_{j_0})$ must be polymorphic, otherwise, $e'@([\sigma_j] \circ [\sigma_k])$ is not well-typed under Δ_1 or Δ_2. Using α-conversion, we can assume that these polymorphic type variables are different from $\Delta_1 \cup \Delta_2$, that is $(\bigcup_{j_0\in J_0} \mathsf{dom}(\sigma_{j_0})) \cap (\Delta_1 \cup \Delta_2) = \emptyset$. So we have $e'@([\sigma_j] \circ [\sigma_k]) \sharp \Delta_1 \cup \Delta_2$. According to Lemma B.8, we have

$$\Delta_1 \cup \Delta_2 \wr \Gamma \wedge \Gamma', (x : t_i(\sigma_j \circ \sigma_k)) \vdash e'@[\sigma_j \circ \sigma_k] : s_i(\sigma_j \circ \sigma_k)$$

for all $i \in I$, $j \in J \cup J'$ and $k \in K$. It is clear that

$$\Delta_1 \cup \Delta_2 = \Delta \cup \Delta' \cup \mathsf{var}(\bigwedge_{i\in I, j\in J\cup J', k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k))$$

By (*abstr*), we have

$$\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J\cup J'}\circ[\sigma_k]_{k\in K}} x.e' : \bigwedge_{i\in I, j\in J\cup J', k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)$$

that is, $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e@[\sigma_j]_{j\in J\cup J'} : t \wedge t'$, where $t = \bigwedge_{i\in I, j\in J, k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)$ and $t' = \bigwedge_{i\in I, j\in J', k\in K} t_i(\sigma_j \circ \sigma_k) \to s_i(\sigma_j \circ \sigma_k)$.
Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = (e_0 \in t\,?\,e_1 : e_2)}$**:** $e@[\sigma_j]_{j\in J} = (e_0@[\sigma_j]_{j\in J} \in t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J})$ and $e@[\sigma_j]_{j\in J'} = (e_0@[\sigma_j]_{j\in J'} \in t\,?\,e_1@[\sigma_j]_{j\in J'} : e_2@[\sigma_j]_{j\in J'})$. Clearly, both typing derivations should end with either (*case*) or (*subsum*). Assume that both derivations end with an instance of (*case*):

$$\dfrac{\dfrac{\ldots}{\Delta \wr \Gamma \vdash e_0@[\sigma_j]_{j\in J} : t_0} \quad \begin{cases} t_0 \not\leq \neg t & \Rightarrow \quad \dfrac{\ldots}{\Delta \wr \Gamma \vdash e_1@[\sigma_j]_{j\in J} : s} \\ t_0 \not\leq t & \Rightarrow \quad \dfrac{\ldots}{\Delta \wr \Gamma \vdash e_2@[\sigma_j]_{j\in J} : s} \end{cases}}{\Delta \wr \Gamma \vdash (e_0@[\sigma_j]_{j\in J} \in t\,?\,e_1@[\sigma_j]_{j\in J} : e_2@[\sigma_j]_{j\in J}) : s}\ (case)$$

$$\dfrac{\dfrac{\ldots}{\Delta' \wr \Gamma' \vdash e_0@[\sigma_j]_{j\in J'} : t_0'} \quad \begin{cases} t_0' \not\leq \neg t & \Rightarrow \quad \dfrac{\ldots}{\Delta' \wr \Gamma' \vdash e_1@[\sigma_j]_{j\in J'} : s'} \\ t_0' \not\leq t & \Rightarrow \quad \dfrac{\ldots}{\Delta' \wr \Gamma' \vdash e_2@[\sigma_j]_{j\in J'} : s'} \end{cases}}{\Delta' \wr \Gamma' \vdash (e_0@[\sigma_j]_{j\in J'} \in t\,?\,e_1@[\sigma_j]_{j\in J'} : e_2@[\sigma_j]_{j\in J'}) : s'}\ (case)$$

By **E**-induction, we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_0@[\sigma_j]_{j\in J\cup J'} : t_0 \wedge t_0'$. Suppose $t_0 \wedge t_0' \not\leq \neg t$, then we must have $t_0 \not\leq \neg t$ and $t_0' \not\leq \neg t$, and the first branch has been checked in both derivations. Therefore we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_1@[\sigma_j]_{j\in J\cup J'} : s \wedge s'$ by the induction hypothesis. Similarly, if $t_0 \wedge t_0' \not\leq t$, we have $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e_2@[\sigma_j]_{j\in J\cup J'} : s \wedge s'$. By applying the rule (*case*), we have

$$\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_0@[\sigma_j]_{j\in J\cup J'} \in t\,?\,e_1@[\sigma_j]_{j\in J\cup J'} : e_2@[\sigma_j]_{j\in J\cup J'}) : s \wedge s'$$

that is, $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e_0 \in t\,?\,e_1 : e_2)@[\sigma_j]_{j\in J\cup J'} : s \wedge s'$.
Otherwise, there exists at least one typing derivation which ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma_i]_{i\in I}}$**:** $e@[\sigma_j]_{j\in J} = e'@([\sigma_j \circ \sigma_i]_{(j,i)\in(J\times I)})$ and $e@[\sigma_j]_{j\in J'} = e'@([\sigma_j \circ \sigma_i]_{(j,i)\in(J'\times I)})$. By **E**-induction on e', we have

$$\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash e'@[\sigma_j \circ \sigma_i]_{(j,i)\in(J\times I)\cup(J'\times I)} : t \wedge t'$$

that is, $\Delta \cup \Delta' \wr \Gamma \wedge \Gamma' \vdash (e'[\sigma_i]_{i\in I})@[\sigma_j]_{j\in J\cup J'} : t \wedge t'$.

□

Corollary B.14. *If* $\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J_1} : t_1$ *and* $\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J_2} : t_2$*, then* $\Delta \wr \Gamma \vdash e@[\sigma_j]_{j\in J_1\cup J_2} : t_1 \wedge t_2$

Proof. Immediate consequence of Lemmas B.13 and B.8. □

B.2 Type soundness

In this section, we prove the soundness of the type system: well-typed expressions do not "go wrong". We proceed in two steps, commonly known as the *subject reduction* and *progress* theorems:

- Subject reduction: a well-typed expression keeps being well-typed during reduction.
- Progress: a well-typed expression can not be "stuck" (*i.e.*, a well-typed expression which is not value can be reduced).

Theorem B.15 (Subject reduction). *Let* e *be an expression and* t *a type. If* $\Delta \wr \Gamma \vdash e : t$ *and* $e \rightsquigarrow e'$*, then* $\Delta \wr \Gamma \vdash e' : t$.

Proof. By induction on the derivation of $\Delta \wr \Gamma \vdash e : t$. We proceed by a case analysis on the last rule used in the derivation of $\Delta \wr \Gamma \vdash e : t$.

(*const*)**:** the expression e is a constant. It cannot be reduced. Thus the result follows.
(*var*)**:** similar to the (*const*) case.
(*pair*)**:** $e = (e_1, e_2), t = t_1 \times t_2$. We have $\Delta \wr \Gamma \vdash e_i : t_i$ for $i = 1..2$. There are two ways to reduce e, that is
(1) $(e_1, e_2) \rightsquigarrow (e_1', e_2)$: by induction, we have $\Delta \wr \Gamma \vdash e_1' : t_1$. Then the rule (*pair*) gives us $\Delta \wr \Gamma \vdash (e_1', e_2) : t_1 \times t_2$.
(2) The case $(e_1, e_2) \rightsquigarrow (e_1, e_2')$ is treated similarly.
(*proj*)**:** $e = \pi_i(e_0), t = t_i, \Delta \wr \Gamma \vdash e_0 : t_1 \times t_2$.
(1) $e_0 \rightsquigarrow e_0'$: $e' = \pi_i(e_0')$. By induction, we have $\Delta \wr \Gamma \vdash e_0' : t_1 \times t_2$. Then the rule (*proj*) gives us $\Delta \wr \Gamma \vdash e' : t_i$.
(2) $e_0 = (v_1, v_2)$: $e' = v_i$. By Lemma B.3, we get $\Delta \wr \Gamma \vdash e' : t_i$.
(*appl*)**:** $e = e_1\ e_2, \Delta \wr \Gamma \vdash e_1 : t \to s$ and $\Delta \wr \Gamma \vdash e_2 : t$.
(1) $e_1\ e_2 \rightsquigarrow e_1'\ e_2$ or $e_1\ e_2 \rightsquigarrow e_1\ e_2'$: similar to the case of (*pair*).
(2) $e_1 = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e_0$, $e_2 = v_2$, $e' = (e_0@[\sigma_j]_{j\in P})\{v_2/x\}$ and $P = \{j \in J \mid \exists i \in I.\ \Delta \wr \Gamma \vdash v_2 : t_i\sigma_j\}$: by Lemma B.3, we have $\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j \leq t \to s$. From the subtyping for arrow types we deduce that $t \leq \bigvee_{i\in I, j\in J} t_i\sigma_j$ and that for any non-empty set $P \subseteq I \times J$ if $t \not\leq \bigvee_{(i,j)\in I\times J\setminus P} t_i\sigma_j$, then $\bigwedge_{(i,j)\in P} s_i\sigma_j \leq s$. Let $P_0 = \{(i,j) \mid \Delta \wr \Gamma \vdash v_2 : t_i\sigma_j\}$. Since $\Delta \wr \Gamma \vdash v_2 : t$ and $t \leq \bigvee_{i\in I, j\in J} t_i\sigma_j$, P_0 is non-empty. Also notice that $t \not\leq \bigvee_{(i,j)\in I\times J\setminus P_0} t_i\sigma_j$, since otherwise there would exist some $(i,j) \notin P_0$ such that $\Delta \wr \Gamma \vdash v_2 : t_i\sigma_j$. As a consequence, we get $\bigwedge_{(i,j)\in P_0} s_i\sigma_j \leq s$. Moreover, since e_1 is well-typed under Δ and Γ, there exists an instance of the rule (*abstr*) which infers a type $\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j$ for e_1 under Δ and Γ and whose premise is $\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash e_0@[\sigma_j] : s_i\sigma_j$ for all $i \in I$ and $j \in J$, where $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j)$. By Lemma B.13, we get $\Delta' \wr \bigwedge_{(i,j)\in P_0}(\Gamma, (x : t_i\sigma_j)) \vdash e_0@[\sigma_j]_{j\in P_0} : \bigwedge_{(i,j)\in P_0} s_i\sigma_j$. Since $\Gamma, (x : \bigwedge_{(i,j)\in P_0} t_i\sigma_j) \simeq \bigwedge_{(i,j)\in P_0}(\Gamma, (x : t_i\sigma_j))$, then from Lemma B.8 we have $\Delta' \wr \Gamma, (x : \bigwedge_{(i,j)\in P_0} t_i\sigma_j) \vdash e_0@[\sigma_j]_{j\in P_0} : \bigwedge_{(i,j)\in P_0} s_i\sigma_j$ and a fortiori $\Delta \wr \Gamma, (x : \bigwedge_{(i,j)\in P_0} t_i\sigma_j) \vdash e_0@[\sigma_j]_{j\in P_0} : \bigwedge_{(i,j)\in P_0} s_i\sigma_j$. Furthermore, by definition of P_0 and the admissibility of the intersection introduction (Lemma B.2) we have that $\Delta \wr \Gamma \vdash v_2 : \bigwedge_{(i,j)\in P_0} t_i\sigma_j$. Thus by Lemma B.6, we get $\Delta \wr \Gamma \vdash e' : \bigwedge_{(i,j)\in P_0} s_i\sigma_j$. Finally, by (*subsum*), we obtain $\Delta \wr \Gamma \vdash e' : s$ as expected.
(*abstr*)**:** $e = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_j]_{j\in J}} x.e_0$. It cannot be reduced. Thus the result follows.
(*case*)**:** $e = e_0 \in s\,?\,e_1 : e_2$.
(1) $e_0 \rightsquigarrow e_0'$ or $e_1 \rightsquigarrow e_1'$ or $e_2 \rightsquigarrow e_2'$: similar to the case of (*pair*).
(2) $e_0 = v_0$ and $\vdash e_0 : s$: we have $e' = e_1$. The typing rule gives us $\Delta \wr \Gamma \vdash e_1 : t$, thus the result follows.
(3) otherwise ($e_0 = v_0$): we have $e' = e_2$. Similar to the above case.
(*inst*)**:** $e = e_1[\sigma], \Delta \wr \Gamma \vdash e_1 : s$, $\sigma \sharp \Delta$ and $e \rightsquigarrow e_1@[\sigma]$. By applying Lemma B.11, we get $\Delta \wr \Gamma \vdash e_1@[\sigma] : s\sigma$.
(*inter*)**:** $e = e_1[\sigma_j]_{j\in J}, \Delta \wr \Gamma \vdash e_1[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j$ and $e \rightsquigarrow e_1@[\sigma_j]_{j\in J}$. By applying Corollary B.12, we get $\Delta \wr \Gamma \vdash e_1@[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j$.

(*subsum*): there exists a type s such that $\Delta \wr \Gamma \vdash e : s \leq t$ and $e \rightsquigarrow e'$. By induction, we have $\Delta \wr \Gamma \vdash e' : s$, then by subsumption we get $\Delta \wr \Gamma \vdash e' : t$.

□

Theorem B.16 (Progress). *Let e be a well-typed closed expression, that is, $\vdash e : t$ for some t. If e is not a value, then there exists an expression e' such that $e \rightsquigarrow e'$.*

Proof. By induction on the derivation of $\vdash e : t$. We proceed by a case analysis of the last rule used in the derivation of $\vdash e : t$.

(*const*): immediate since a constant is a value.
(*var*): impossible since a variable cannot be well-typed in an empty environment.
(*pair*): $e = (e_1, e_2)$, $t = t_1 \times t_2$, and $\vdash e_i : t_i$ for $i = 1..2$. If one of the e_i can be reduced, then e can also be reduced. Otherwise, by induction, both e_1 and e_2 are values, and so is e.
(*proj*): $e = \pi_i(e_0)$, $t = t_i$, and $\vdash e_0 : t_1 \times t_2$. If e_0 can be reduced to e_0', then $e \rightsquigarrow \pi_i(e_0')$. Otherwise, e_0 is a value. By Lemma B.3, we get $e_0 = (v_1, v_2)$, and thus $e \rightsquigarrow v_i$.
(*appl*): $e = e_1\ e_2$, $\vdash e_1 : t \to s$ and $\vdash e_2 : t$. If one of the e_i can be reduced, then e can also be reduced. Otherwise, by induction, both e_1 and e_2 are values. By Lemma B.3, we get $e_1 = \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$ such that $\bigwedge_{i\in I, j\in J} t_i\sigma_j \to s_i\sigma_j \leq t \to s$. By the definition of subtyping for arrow types, we have $t \leq \bigvee_{i\in I, j\in J} t_i\sigma_j$. Moreover, as $\vdash e_2 : t$, the set $P = \{j \in J \mid \exists i \in I.\ \vdash e_2 : t_i\sigma_j\}$ is non-empty. Then $e \rightsquigarrow (e_0@[\sigma_j]_{j\in P})\{e_2/x\}$.
(*abstr*): the expression e is an abstraction which is well-typed under the empty environment. It is thus a value.
(*case*): $e = e_0 \in s\ ?\ e_1 : e_2$. If e_0 can be reduced, then e can also be reduced. Otherwise, by induction, e_0 is a value v. If $\vdash v : s$, then we have $e \rightsquigarrow e_1$. Otherwise, $e \rightsquigarrow e_2$.
(*inst*): $e = e_1[\sigma]$, $t = s\sigma$ and $\vdash e_1 : s$. Then $e \rightsquigarrow e_1@[\sigma]$.
(*inter*): $e = e_1[\sigma_j]_{j\in J}$, $t = \bigwedge_{j\in J} t_j$ and $\vdash e_1[\sigma_j] : t_j$ for all $j \in J$. It is clear that $e \rightsquigarrow e_1@[\sigma_j]_{j\in J}$.
(*subsum*): straightforward application of the induction hypothesis.

□

We now conclude that the type system is type sound.

Corollary B.17 (Type soundness). *Let e be a well-typed closed expression, that is, $\vdash e : t$ for some t. Then either e diverges or it returns a value of type t.*

Proof. Consequence of Theorems B.16 and B.15. □

B.3 Expressing intersection types

We now prove that the calculus with explicit substitutions is able to derive the same typings as the Barendregt, Coppo, Dezani (BCD) intersection type system [1] without the universal type ω. We remind the BCD types (a strict subset of $\mathscr{T}$), the BCD typing rules (without ω) and subtyping relation in Figure 5, where we use m to range over pure λ-calculus expressions. To make the correspondence between the systems easier, we adopt a n-ary version of the intersection typing rule. Henceforth, we use D to range over BCD typing derivations. We first remark that the BCD subtyping relation is included in the one of this work.

Lemma B.18. *If $t_1 \leq_{BCD} t_2$ then $t_1 \leq t_2$.*

Proof. All the BCD subtyping rules are admissible in [12] and, a fortiori, in our type system. □

In this subsection, we restrict the grammar of expressions with explicit substitutions to

$$e ::= x \mid e\ e \mid \lambda^{\bigwedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \tag{24}$$

and we write $\lceil e \rceil$ for the pure λ-calculus expression obtained by removing all types references (i.e., interfaces and decorations) from e. Given a pure λ-calculus expression m and a derivation of a judgement $\Gamma \vdash_{BCD} m : t$, we build an expression e such that $\lceil e \rceil = m$ and $\Delta \wr \Gamma \vdash e : t$ for some set Δ of type variables. With the restricted grammar of (24), the intersection typing rule is used only in conjunction with the abstraction typing rule. We prove that it is possible to put a similar restriction on derivations of BCD typing judgements.

Types:

$$t ::= \alpha \mid t \to t \mid t \wedge t$$

Typing rules:

$$\frac{}{\Gamma \vdash_{BCD} x : \Gamma(x)}\ (\textit{BCD var}) \qquad \frac{\Gamma \vdash_{BCD} m_1 : t_1 \to t_2 \qquad \Gamma \vdash m_2 : t_1}{\Gamma \vdash_{BCD} m_1\ m_2 : t_2}\ (\textit{BCD app})$$

$$\frac{\Gamma, x : t_1 \vdash_{BCD} m : t_2}{\Gamma \vdash_{BCD} \lambda x.m : t_1 \to t_2}\ (\textit{BCD abstr}) \qquad \frac{\Gamma \vdash_{BCD} m : t_i}{\Gamma \vdash_{BCD} m : \bigwedge_{i \in I} t_i}\ \begin{array}{l} i \in I \\ |I| > 1 \end{array}\ (\textit{BCD inter})$$

$$\frac{\Gamma \vdash_{BCD} m : t_1 \qquad t_1 \leq_{BCD} t_2}{\Gamma \vdash m : t_2}\ (\textit{BCD sub})$$

Subtyping relation:

$$\frac{}{t \leq_{BCD} t} \qquad \frac{}{t \leq_{BCD} t \wedge t} \qquad \frac{}{t_1 \wedge t_2 \leq_{BCD} t_1} \qquad \frac{}{(t_1 \to t_2) \wedge (t_1 \to t_3) \leq_{BCD} t_1 \to (t_2 \wedge t_3)}$$

$$\frac{t_1 \leq_{BCD} t_2 \qquad t_2 \leq_{BCD} t_3}{t_1 \leq_{BCD} t_3} \qquad \frac{t_1 \leq_{BCD} t_3 \qquad t_2 \leq_{BCD} t_4}{t_1 \wedge t_2 \leq_{BCD} t_3 \wedge t_4} \qquad \frac{t_3 \leq_{BCD} t_1 \qquad t_2 \leq_{BCD} t_4}{t_1 \to t_2 \leq_{BCD} t_3 \to t_4}$$

Figure 5. The BCD type system

Definition B.19. *Let D be a BCD typing derivation. We say D is in* intersection-abstraction normal form *if* (*BCD inter*) *is used only immediately after* (*BCD abstr*) *in D, that is to say, all uses of* (*BCD inter*) *in D are of the form*

$$\frac{\dfrac{\dfrac{D_i}{\Gamma, x : t_i \vdash_{BCD} m : s_i}}{\Gamma \vdash_{BCD} \lambda x.m : t_i \to s_i} i \in I}{\Gamma \vdash_{BCD} \lambda x.m : \bigwedge_{i \in I} t_i \to s_i}$$

Definition B.20. *Let D be a (BCD) typing derivation. We define the* size *of D, denoted by $|D|$, as the number of rules used in D.*

We prove that any BCD typing judgement can be proved with a derivation in intersection-abstraction normal form.

Lemma B.21. *If $\Gamma \vdash_{BCD} m : t$, then there exists a derivation in intersection-abstraction normal form proving this judgement.*

Proof. Let D be the derivation proving $\Gamma \vdash_{BCD} m : t$. We proceed by induction on the size of D. If $|D| = 1$ then the rule (*BCD var*) has been used, and D is in intersection-abstraction normal form. Otherwise, assume that $|D| > 1$. We proceed by case analysis on the last rule used in D.

(*BCD sub*)**:** D ends with (*BCD sub*):

$$D = \frac{D' \qquad t' \leq t}{\Gamma \vdash_{BCD} m : t}$$

where D' proves a judgement $\Gamma \vdash_{BCD} m : t'$. By the induction hypothesis, there exists a derivation D'' in intersection-abstraction normal form which proves the same judgement as D'. Then

$$\frac{D'' \qquad t' \leq t}{\Gamma \vdash_{BCD} m : t}$$

is in intersection-abstraction normal form and proves the same judgement as D.

(*BCD abstr*)**:** similar to the case of (*BCD sub*).

(*BCD app*)**:** similar to the case of (*BCD sub*).

(*BCD inter*)**:** D ends with (*BCD inter*):

$$D = \frac{D_i}{\Gamma \vdash_{BCD} m : t} i \in I$$

where each D_i proves a judgement $\Gamma \vdash_{BCD} m : t_i$ and $t = \bigwedge_{i \in I} t_i$. We distinguish several cases.

If one of the derivations ends with (*BCD sub*)**,** there exists $i_0 \in I$ such that

$$D_{i_0} = \frac{D'_{i_0} \qquad t'_{i_0} \leq t_{i_0}}{\Gamma \vdash_{BCD} m : t_{i_0}}$$

The derivation

$$D' = \frac{D_i \qquad D'_{i_0}}{\Gamma \vdash_{BCD} m : \bigwedge_{i \in I \setminus \{i_0\}} t_i \wedge t'_{i_0}} i \in I \setminus \{i_0\}$$

is smaller than D, so by the induction hypothesis, there exists D'' in intersection-abstraction normal form which proves the same judgement as D'. Then the derivation

$$\frac{D'' \qquad \bigwedge_{i \in I \setminus \{i_0\}} t_i \wedge t'_{i_0} \leq_{BCD} \bigwedge_{i \in I} t_i}{\Gamma \vdash_{BCD} m : t}$$

is in intersection-abstraction normal form, and proves the same judgement as D.

If one of the derivations ends with (*BCD inter*)**,** there exists $i_0 \in I$ such that

$$D_{i_0} = \frac{D_{j,i_0}}{\Gamma \vdash_{BCD} m : \bigwedge_{j \in J} t_{j,i_0}} j \in J$$

with $t_{i_0} = \bigwedge_{j \in J} t_{j,i_0}$. The derivation

$$D' = \frac{D_i \qquad D_{j,i_0} \quad i \in I \setminus \{i_0\} \quad j \in J}{\Gamma \vdash_{BCD} m : t}$$

is smaller than D, so by the induction hypothesis, there exists D'' in intersection-abstraction normal form which proves the same judgement as D', which is the same as the judgement of D.

If all the derivations are uses of (*BCD var*)**,** then for all $i \in I$, we have

$$D_i = \frac{}{\Gamma \vdash_{BCD} x : \Gamma(x)}$$

which implies $t = \bigwedge_{i \in I} \Gamma(x)$ and $m = x$. Then the derivation

$$\frac{\dfrac{}{\Gamma \vdash_{BCD} x : \Gamma(x)} \qquad \Gamma(x) \leq_{BCD} t}{\Gamma \vdash_{BCD} x : t}$$

is in intersection-abstraction normal form and proves the same judgement as D.

If all the derivations end with (*BCD app*)**,** then for all $i \in I$, we have

$$D_i = \frac{D_i^1 \qquad D_i^2}{\Gamma \vdash_{BCD} m_1 \; m_2 : t_i}$$

where $m = m_1 \; m_2$, D_i^1 proves $\Gamma \vdash_{BCD} m_1 : s_i \to t_i$, and D_i^2 proves $\Gamma \vdash_{BCD} m_2 : s_i$ for some s_i. Let

$$D_1 = \frac{D_i^1}{\Gamma \vdash_{BCD} m_1 : \bigwedge_{i \in I} s_i \to t_i} i \in I \qquad\qquad D_2 = \frac{D_i^2}{\Gamma \vdash_{BCD} m_2 : \bigwedge_{i \in I} s_i} i \in I$$

Both D_1 and D_2 are smaller than D, so by the induction hypothesis, there exist D'_1, D'_2 in intersection-abstraction normal form which prove the same judgements as D_1 and D_2 respectively. Then the derivation

$$\frac{\dfrac{D'_1 \qquad \bigwedge_{i \in I} s_i \to t_i \leq_{BCD} (\bigwedge_{i \in I} s_i) \to (\bigwedge_{i \in I} t_i)}{\Gamma \vdash_{BCD} m_1 : (\bigwedge_{i \in I} s_i) \to (\bigwedge_{i \in I} t_i)} \qquad D'_2}{\Gamma \vdash m_1 \; m_2 : t}$$

is in intersection-abstraction normal form and proves the same derivation as D.

If all the derivations end with (*BCD abstr*)**,** then for all $i \in I$, we have

$$D_i = \frac{D'_i}{\Gamma \vdash_{BCD} \lambda x.m' : t_i} \; (\textit{BCD abstr})$$

with $m = \lambda x.m'$. For all $i \in I$, D'_i is smaller than D, so by induction there exists D''_i in intersection-abstraction normal form which proves the same judgement as D'_i. Then the derivation

$$\frac{\dfrac{D''_i}{\Gamma \vdash_{BCD} \lambda x.m' : t_i}}{\Gamma \vdash_{BCD} \lambda x.m' : \bigwedge_{i \in I} t_i} i \in I$$

is in intersection-abstraction normal form, and proves the same judgement as D.

□

We now sketch the principles behind the construction of e from D in intersection-abstraction normal form. If D proves a judgement $\Gamma \vdash_{BCD} \lambda x.m : t \to s$, without any top-level intersection, then we simply put $t \to s$ in the interface of the corresponding expression $\lambda^{t\to s} x.e$.

For a judgement $\Gamma \vdash_{BCD} \lambda x.m : \bigwedge_{i\in I} t_i \to s_i$, we build an expression $\lambda^{\alpha\to\beta}_{[\sigma_i]_{i\in I}} x.e$ where each σ_i corresponds to the derivation which types $\lambda x.m$ with $t_i \to s_i$. For example, let $m = \lambda f.\lambda x.f\ x$, and consider the judgement $\vdash_{BCD} m : ((t_1 \to t_2) \to t_1 \to t_2) \wedge ((s_1 \to s_2) \to s_1 \to s_2)$. We first annotate each abstraction in m with types $\alpha_j \to \beta_j$, where α_j, β_j are fresh, distinct variables, giving us $e = \lambda^{\alpha_1\to\beta_1} f.\lambda^{\alpha_2\to\beta_2} x.f\ x$. Comparing $\lambda^{\alpha_2\to\beta_2} x.f\ x$ to the judgement $f : t_1 \to t_2 \vdash_{BCD} \lambda x.f\ x : t_1 \to t_2$ and e to $\vdash_{BCD} m : (t_1 \to t_2) \to t_1 \to t_2$, we compute $\sigma_1 = \{t_1 \to t_2/\alpha_1, t_1 \to t_2/\beta_1, t_1/\alpha_2, t_2/\beta_2\}$. We compute similarly σ_2 from the derivation of $\vdash_{BCD} m : (s_1 \to s_2) \to s_1 \to s_2$, and we obtain finally

$$\vdash \lambda^{\alpha_1\to\beta_1}_{[\sigma_1,\sigma_2]} f.\lambda^{\alpha_2\to\beta_2} x.f\ x : ((t_1 \to t_2) \to t_1 \to t_2) \wedge ((s_1 \to s_2) \to s_1 \to s_2)$$

as wished.

The problem becomes more complex when we have nested uses of the intersection typing rule. For example, let $m = \lambda f.\lambda g.g\ (\lambda x.f\ (\lambda y.x\ y))$ and consider the judgement $\vdash_{BCD} m : (t_f \to t_g \to t_4) \wedge (s_f \to s_g \to s_7)$ with

$$\begin{aligned} t_f &= (t_1 \to t_2) \to t_3 \\ t_g &= t_f \to t_4 \\ s_f &= ((s_1 \to s_2) \to s_3) \wedge ((s_4 \to s_5) \to s_6) \\ s_g &= s_f \to s_7 \end{aligned}$$

Notice that, to derive $\vdash_{BCD} m : s_f \to s_g \to s_7$, we have to prove $f : s_f, g : s_g \vdash_{BCD} \lambda x.f\ (\lambda y.x\ y) : s_f$, which requires the *(BCD inter)* rule. As before, we annotate m with fresh interfaces, obtaining $\lambda^{\alpha_1\to\beta_1} f.\lambda^{\alpha_2\to\beta_2} g.g\ (\lambda^{\alpha_3\to\beta_3} x.f\ (\lambda^{\alpha_4\to\beta_4} y.x\ y))$. Because the intersection typing rule is used twice (once to type m, and once to type $m' = \lambda x.f\ (\lambda y.x\ y)$), we want to compute four substitutions $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ to obtain a decorated expression $\lambda^{\alpha_1\to\beta_1}_{[\sigma_1,\sigma_2]} f.\lambda^{\alpha_2\to\beta_2} g.g\ (\lambda^{\alpha_3\to\beta_3}_{[\sigma_3,\sigma_4]} x.f\ (\lambda^{\alpha_4\to\beta_4} y.x\ y))$. The difficult part is in computing σ_3 and σ_4; in one case (corresponding to the branch $\vdash_{BCD} m : t_f \to t_g \to t_4$), we want $\sigma_3 = \sigma_4 = \{t_1 \to t_2/\alpha_3, t_3/\beta_3, t_1/\alpha_4, t_2/\beta_4\}$ to obtain $f : t_f, g : t_g \vdash \lambda^{\alpha_3\to\beta_3}_{[\sigma_3,\sigma_4]} x.f\ (\lambda^{\alpha_4\to\beta_4} y.x\ y) : t_f$, and in the other case (corresponding to the derivation $\vdash_{BCD} m : s_f \to s_g \to s_7$), we want $\sigma_3 = \{s_1 \to s_2/\alpha_3, s_3/\beta_3, s_1/\alpha_4, s_2/\beta_4\}$ and $\sigma_4 = \{s_4 \to s_5/\alpha_3, s_6/\beta_3, s_4/\alpha_4, s_5/\beta_4\}$ to obtain $f : s_f, g : s_g \vdash \lambda^{\alpha_3\to\beta_3}_{[\sigma_3,\sigma_4]} x.f\ (\lambda^{\alpha_4\to\beta_4} y.x\ y) : s_f$. To resolve this issue, we use intermediate fresh variables $\alpha'_3, \beta'_3, \alpha'_4, \beta'_4$ and $\alpha''_3, \beta''_3, \alpha''_4, \beta''_4$ in the definition of σ_3 and σ_4. We define

$$\begin{aligned} \sigma_1 =& \{t_f/\alpha_1, t_g \to t_4/\beta_1, t_g/\alpha_2, t_4/\beta_2, t_1 \to t_2/\alpha'_3, t_3/\beta'_3, t_1/\alpha'_4, t_2/\beta'_4, \\ & t_1 \to t_2/\alpha''_3, t_3/\beta''_3, t_1/\alpha''_4, t_2/\beta''_4\} \\ \sigma_2 =& \{s_f/\alpha_1, s_g \to s_7/\beta_1, s_g/\alpha_2, s_7/\beta_2, s_1 \to s_2/\alpha'_3, s_3/\beta'_3, s_1/\alpha'_4, s_2/\beta'_4, \\ & s_4 \to s_5/\alpha''_3, s_6/\beta''_3, s_4/\alpha''_4, s_5/\beta''_4\} \\ \sigma_3 =& \{\alpha'_3/\alpha_3, \beta'_3/\beta_3, \alpha'_4/\alpha_4, \beta'_4/\beta_4\} \\ \sigma_4 =& \{\alpha''_3/\alpha_3, \beta''_3/\beta_3, \alpha''_4/\alpha_4, \beta''_4/\beta_4\} \end{aligned}$$

Because the substitutions compose themselves, we obtain

$$\vdash \lambda^{\alpha_1\to\beta_1}_{[\sigma_1,\sigma_2]} f.\lambda^{\alpha_2\to\beta_2} g.g\ (\lambda^{\alpha_3\to\beta_3}_{[\sigma_3,\sigma_4]} x.f\ (\lambda^{\alpha_4\to\beta_4} y.x\ y)) : (t_f \to t_g \to t_4) \wedge (s_f \to s_g \to s_7)$$

as wished.

In the next lemma, given n derivations $D_1, \ldots, D_n$ in intersection-abstraction normal form for a same expression m, we construct an expression e containing fresh interfaces and decorations with fresh variables if needed (as explained in the above example) and n substitutions $\sigma_1, \ldots, \sigma_n$ corresponding to $D_1, \ldots, D_n$. Let $\mathsf{var}(D_1, \ldots, D_n)$ denote the set of type variables occurring in the types in $D_1, \ldots, D_n$.

Lemma B.22. *Let m be a pure λ-calculus expression, Δ be a set of type variables, and $D_1, \ldots, D_n$ be derivations in intersection-abstraction normal form such that D_i proves $\Gamma_i \vdash_{BCD} m : t_i$ for all i. Let Δ' be a set of type variables such that $\mathsf{var}(D_1, \ldots, D_n) \subseteq \Delta'$. There exist e, $\sigma_1, \ldots, \sigma_n$ such that $\lceil e \rceil = m$, $\mathsf{dom}(\sigma_1) = \ldots = \mathsf{dom}(\sigma_n) \subseteq \mathsf{tv}(e)$, $\mathsf{tv}(e) \cap (\Delta \cup \Delta') = \emptyset$, and $\Delta' \S \Gamma_i \vdash e@[\sigma_i] : t_i$ for all i.*

Proof. We proceed by induction on the sum of the size of $D_1, \ldots, D_n$. If this sum is equal to n, then each D_i is a use of the *(BCD var)* rule, and we have $m = x$ for some x. Let $e = x$ and σ_i be the identity; we can then easily check that the result holds. Otherwise, assume this sum is strictly greater than n. We proceed by case analysis on $D_1, \ldots, D_n$.

Case 1: If one of the derivations ends with (*BCD sub*), there exists i_0 such that

$$D_{i_0} = \frac{D'_{i_0} \qquad t'_{i_0} \leq_{BCD} t_{i_0}}{\Gamma_{i_0} \vdash_{BCD} m : t_{i_0}}$$

Clearly, the sum of the size of $D_1, \ldots, D'_{i_0}, \ldots, D_n$ is smaller than that of $D_1, \ldots, D_n$, and $\mathsf{var}(D_1, \ldots, D'_{i_0}, \ldots, D_n) \subseteq \Delta'$. So by the induction hypothesis, we have

$$\begin{aligned} \exists e, \sigma_1, \ldots, \sigma_n. \quad & \lceil e \rceil = m \\ & \text{and } \mathsf{dom}(\sigma_1) = \ldots = \mathsf{dom}(\sigma_n) \subseteq \mathsf{tv}(e) \\ & \text{and } \mathsf{tv}(e) \cap (\Delta \cup \Delta') = \emptyset \\ & \text{and } \forall i \in \{1, \ldots, n\} \setminus \{i_0\}.\ \Delta' \wr \Gamma_i \vdash e@[\sigma_i] : t_i \\ & \text{and } \Delta' \wr \Gamma_{i_0} \vdash e@[\sigma_{i_0}] : t'_{i_0}. \end{aligned}$$

Since $t'_{i_0} \leq_{BCD} t_{i_0}$, by Lemma B.18, we have $t'_{i_0} \leq t_{i_0}$. Therefore $\Delta' \wr \Gamma_{i_0} \vdash e@[\sigma_{i_0}] : t_{i_0}$ holds, and for all i, we have $\Delta' \wr \Gamma_i \vdash e@[\sigma_i] : t_i$ as wished.

Case 2: If all the derivations end with (*BCD app*), then we have $m = m_1\ m_2$, and for all i

$$D_i = \frac{D_i^1 \qquad D_i^2}{\Gamma_i \vdash_{BCD} m_1\ m_2 : t_i}$$

where D_i^1 proves $\Gamma_i \vdash_{BCD} m_1 : s_i \to t_i$ and D_i^2 proves $\Gamma_i \vdash_{BCD} m_2 : s_i$ for some s_i. Applying the induction hypothesis on $D_1^1, \ldots, D_n^1$ (with Δ and Δ'), we have

$$\begin{aligned} \exists e_1, \sigma_1^1, \ldots, \sigma_n^1. \quad & \lceil e \rceil_1 = m_1 \\ & \text{and } \mathsf{dom}(\sigma_1^1) = \ldots = \mathsf{dom}(\sigma_n^1) \subseteq \mathsf{tv}(e_1) \\ & \text{and } \mathsf{tv}(e_1) \cap (\Delta \cup \Delta') = \emptyset \\ & \text{and } \forall i \in \{1, \ldots, n\}.\ \Delta' \wr \Gamma_i \vdash e_1@[\sigma_i^1] : s_i \to t_i. \end{aligned}$$

Similarly, by induction on $D_1^2, \ldots, D_n^2$ (with $\Delta \cup \mathsf{tv}(e_1)$ and Δ'),

$$\begin{aligned} \exists e_2, \sigma_1^2, \ldots, \sigma_n^2. \quad & \lceil e \rceil_2 = m_2 \\ & \text{and } \mathsf{dom}(\sigma_1^2) = \ldots = \mathsf{dom}(\sigma_n^2) \subseteq \mathsf{tv}(e_2) \\ & \text{and } \mathsf{tv}(e_2) \cap (\Delta \cup \mathsf{tv}(e_1) \cup \Delta') = \emptyset \\ & \text{and } \forall i \in \{1, \ldots, n\}.\ \Delta' \wr \Gamma_i \vdash e_2@[\sigma_i^2] : s_i. \end{aligned}$$

From $\mathsf{tv}(e_2) \cap (\Delta \cup \mathsf{tv}(e_1) \cup \Delta') = \emptyset$, we deduce $\mathsf{tv}(e_1) \cap \mathsf{tv}(e_2) = \emptyset$. Let $i \in \{1, \ldots, n\}$. Because $\mathsf{dom}(\sigma_i^1) \subseteq \mathsf{tv}(e_1)$ and $\mathsf{dom}(\sigma_i^2) \subseteq \mathsf{tv}(e_2)$, we have $\mathsf{dom}(\sigma_i^1) \cap \mathsf{dom}(\sigma_i^2) = \emptyset$, $\mathsf{dom}(\sigma_i^1) \cap \mathsf{tv}(e_2) = \emptyset$, and $\mathsf{dom}(\sigma_i^2) \cap \mathsf{tv}(e_1) = \emptyset$. Consequently, by Lemma B.9, we have $\Delta' \wr \Gamma_i \vdash e_1@[\sigma_i^1 \cup \sigma_i^2] : s_i \to t_i$ and $\Delta' \wr \Gamma_i \vdash e_2@[\sigma_i^1 \cup \sigma_i^2] : s_i$. Therefore, we have $\Delta' \wr \Gamma_i \vdash (e_1\ e_2)@[\sigma_i^1 \cup \sigma_i^2] : t_i$. So we have the required result with $e = e_1\ e_2$ and $\sigma_i = \sigma_i^1 \cup \sigma_i^2$.

Case 3: If all the derivations end with (*BCD abstr*), then $m = \lambda x.m_1$, and for all i,

$$D_i = \frac{D'_i}{\Gamma_i \vdash_{BCD} m : t_i}$$

where D'_i proves $\Gamma_i, x : t_i^1 \vdash_{BCD} m_1 : t_i^2$ and $t_i = t_i^1 \to t_i^2$. By the induction hypothesis,

$$\begin{aligned} \exists e_1, \sigma'_1, \ldots, \sigma'_n. \quad & \lceil e \rceil_1 = m_1 \\ & \text{and } \mathsf{dom}(\sigma'_1) = \ldots = \mathsf{dom}(\sigma'_n) \subseteq \mathsf{tv}(e_1) \\ & \text{and } \mathsf{tv}(e_1) \cap (\Delta \cup \Delta') = \emptyset \\ & \text{and } \forall i \in \{1, \ldots, n\}.\ \Delta' \wr \Gamma_i, x : t_i^1 \vdash e_1@[\sigma'_i] : t_i^2. \end{aligned}$$

Let α, β be two fresh type variables. So $\{\alpha, \beta\} \cap (\Delta \cup \Delta') = \emptyset$ and $\{\alpha, \beta\} \cap \mathsf{tv}(e_1) = \emptyset$. Take $i \in \{1, \ldots, n\}$. Let $\sigma_i = \{t_i^1/\alpha, t_i^2/\beta\} \cup \sigma'_i$, and $e = \lambda^{\alpha \to \beta} x.e_1$. We have $\mathsf{dom}(\sigma_i) = \{\alpha, \beta\} \cup \mathsf{dom}(\sigma'_i) \subseteq \{\alpha, \beta\} \cup \mathsf{tv}(e_1) = \mathsf{tv}(e)$. Besides, we have $\mathsf{tv}(e) \cap (\Delta \cup \Delta') = \emptyset$. Because $\mathsf{tv}(e_1) \cap \{\alpha, \beta\} = \emptyset$, we have $\mathsf{dom}(\sigma'_i) \cap \{\alpha, \beta\} = \emptyset$, and $\Delta' \wr \Gamma_i, x : t_i^1 \vdash e_1@[\sigma_i] : t_i^2$ by Lemma B.9, which is equivalent to $\Delta' \wr \Gamma_i, x : \alpha\sigma_i \vdash e_1@[\sigma_i] : \beta\sigma_i$. Because $\Delta' \cup \mathsf{var}(t_i^1 \to t_i^2) = \Delta'$, by the abstraction rule, we have $\Delta' \wr \Gamma_i \vdash \lambda_{[\sigma_i]}^{\alpha \to \beta} x.e_1 : t_i$, *i.e.*, $\Delta' \wr \Gamma_i \vdash (\lambda^{\alpha \to \beta}.e_1)@[\sigma_i] : t_i$. Therefore, we have the required result.

Case 4: If one of the derivations ends with (*BCD inter*), then $m = \lambda x.m_1$. The derivations end with either (*BCD inter*) or (*BCD abstr*) (we omit the already treated case of (*BCD sub*)). For simplicity, we suppose they all end with (*BCD inter*), as it is the same if some of them end with (*BCD abstr*) (note that Case 3 is a special case of Case 4). For all i, we have

$$D_i = \frac{\dfrac{D_i^j}{\Gamma_i \vdash_{BCD} m : s_i^j \to t_i^j}\, j \in J_i}{\Gamma_i \vdash_{BCD} m : \bigwedge_{j \in J_i} s_i^j \to t_i^j}$$

where D_i^j proves $\Gamma_i, x : s_i^j \vdash_{BCD} m_1 : t_i^j$ for all $j \in J_i$ and $t_i = \bigwedge_{j\in J_i} s_i^j \to t_i^j$ for all i. By the induction hypothesis on the sequence of D_i^j,

$$\begin{aligned}\exists e_1, (\sigma_1^j)_{j\in J_1}, \dots, (\sigma_n^j)_{j\in J_n}. \quad & \lceil e_1 \rceil = m_1 \\ & \text{and } \forall i, i', j, j'.\ \mathsf{dom}(\sigma_i^j) = \mathsf{dom}(\sigma_{i'}^{j'}) \text{ and } \mathsf{dom}(\sigma_i^j) \subseteq \mathsf{tv}(e_1) \\ & \text{and } \mathsf{tv}(e_1) \cap (\Delta \cup \Delta') = \emptyset \\ & \text{and } \forall i, j.\ \Delta' \fatsemi \Gamma_i, x : s_i^j \vdash e_1@[\sigma_i^j] : t_i^j.\end{aligned}$$

Let $p = max_{i\in\{1,\dots,n\}}\{|J_i|\}$. For all i, we complete the sequence of substitutions (σ_i^j) so that it contains exactly p elements, by repeating the last one $p - |J_i|$ times, and we number them from 1 to p. All the σ_i^j have the same domain (included in $\mathsf{tv}(e_1)$), that we number from 1 to q. Then $\sigma_i^j = \bigcup_{k\in\{1,\dots,q\}}\{t_{i,j}^k/\alpha_k\}$ for some types $(t_{i,j}^k)$. Let $\alpha, \beta, (\alpha_{j,k})_{j\in\{1,\dots,p\},k\in\{1,\dots,q\}}, (\alpha_{j,0}, \beta_{j,0})_{j\in\{1,\dots,p\}}$ be fresh pairwise distinct variables (which do not occur in $\Delta \cup \Delta' \cup \mathsf{tv}(e_1)$). For all $j \in \{1, \dots, p\}$, $i \in \{1, \dots, n\}$, we define:

$$\begin{aligned}\sigma_j &= \bigcup_{k\in\{1,\dots,q\}} \{\alpha_{j,k}/\alpha_k\} \cup \{\alpha_{j,0}/\alpha, \beta_{j,0}/\beta\} \\ e &= \lambda^{\alpha\to\beta}_{[\sigma_j]_{j\in\{1,\dots,p\}}} x.e_1 \\ \sigma_i &= \bigcup_{j\in\{1,\dots,p\},k\in\{1,\dots,q\}} \{t_{i,j}^k/\alpha_{j,k}\} \cup \bigcup_{j\in\{1,\dots,p\}} \{s_i^j/\alpha_{j,0}, t_i^j/\beta_{j,0}\}\end{aligned}$$

For all i, j, k, we have by construction $(\alpha_k\sigma_j)\sigma_i = \alpha_k\sigma_i^j$, $(\alpha\sigma_j)\sigma_i = s_i^j$, and $(\beta\sigma_j)\sigma_i = t_i^j$. Moreover, since

$$\begin{aligned}\mathsf{tv}(e) &= \mathsf{tv}(e_1@[\sigma_j]_{j\in\{1,\dots,p\}}) \cup \textstyle\bigcup_{j\in\{1,\dots,p\}} \mathsf{var}((\alpha \to \beta)\sigma_j) \\ &= (\mathsf{tv}(e_1))[\sigma_j]_{j\in\{1,\dots,p\}} \cup \{\alpha_{j,0}, \beta_{j,0}\}_{j\in\{1,\dots,p\}} \\ &\supseteq (\mathsf{dom}(\sigma_i^j))[\sigma_j]_{j\in\{1,\dots,p\}} \cup \{\alpha_{j,0}, \beta_{j,0}\}_{j\in\{1,\dots,p\}} \\ &= (\{\alpha_k\}_{k\in\{1,\dots,q\}})[\sigma_j]_{j\in\{1,\dots,p\}} \cup \{\alpha_{j,0}, \beta_{j,0}\}_{j\in\{1,\dots,p\}} \\ &= \{\alpha_{j,k}, \beta_{j,k}\}_{j\in\{1,\dots,p\},k\in\{1,\dots,q\}} \cup \{\alpha_{j,0}, \beta_{j,0}\}_{j\in\{1,\dots,p\}}\end{aligned}$$

and

$$\begin{aligned}\mathsf{tv}(e) &= (\mathsf{tv}(e_1))[\sigma_j]_{j\in\{1,\dots,p\}} \cup \{\alpha_{j,0}, \beta_{j,0}\}_{j\in\{1,\dots,p\}} \\ &\subseteq \mathsf{tv}(e_1) \cup \textstyle\bigcup_{j\in\{1,\dots,p\}} \mathsf{tvran}(\sigma_j) \cup \{\alpha_{j,0}, \beta_{j,0}\}_{j\in\{1,\dots,p\}} \\ &= \mathsf{tv}(e_1) \cup \{\alpha_{j,k}, \beta_{j,k}\}_{j\in\{1,\dots,p\},k\in\{1,\dots,q\}} \cup \{\alpha_{j,0}, \beta_{j,0}\}_{j\in\{1,\dots,p\}}\end{aligned}$$

we have $\mathsf{dom}(\sigma_i) \subseteq \mathsf{tv}(e)$ and $\mathsf{tv}(e) \cap (\Delta \cup \Delta') = \emptyset$. Because $\Delta' \fatsemi \Gamma_i, x : s_i^j \vdash e_1@[\sigma_i^j] : t_i^j$, by Lemma B.9, we have $\Delta' \fatsemi \Gamma_i, x : s_i^j \vdash e_1@[\sigma_i \circ \sigma_j] : t_i^j$, which is equivalent to $\Delta' \fatsemi \Gamma_i, x : \alpha(\sigma_i \circ \sigma_j) \vdash e_1@[\sigma_i \circ \sigma_j] : \beta(\sigma_i \circ \sigma_j)$ for all i, j. Since $\Delta' \cup \bigcup_{j\in\{1,\dots,p\}} \mathsf{var}(s_i^j \to t_i^j) = \Delta'$, for a given i, by the abstraction typing rule we have $\Delta' \fatsemi \Gamma_i \vdash \lambda^{\alpha\to\beta}_{[\sigma_i\circ\sigma_j]_{j\in\{1,\dots,p\}}} x.e_1 : \bigwedge_{j\in\{1,\dots,p\}} s_i^j \to t_i^j \leq \bigwedge_{j\in J_i} s_i^j \to t_i^j = t_i$. This is equivalent to $\Delta' \fatsemi \Gamma_i \vdash \lambda^{\alpha\to\beta}_{[\sigma_j]_{j\in\{1,\dots,p\}}} x.e_1@[\sigma_i] : t_i$, hence $\Delta' \fatsemi \Gamma_i \vdash e@[\sigma_i] : t_i$ holds for all i, as wished.

□

We are now ready to prove the main result of this subsection.

Theorem B.23. *If $\Gamma \vdash_{BCD} m : t$, then there exist e, Δ such that $\Delta \fatsemi \Gamma \vdash e : t$ and $\lceil e \rceil = m$.*

Proof. By Lemma B.21, there exists D in intersection-abstraction normal form such that D proves $\Gamma \vdash_{BCD} m : t$. Let Δ be a set of type variables such that $\mathsf{var}(D) \subseteq \Delta$. We prove by induction on $|D|$ that there exists e such that $\Delta \fatsemi \Gamma \vdash e : t$ and $\lceil e \rceil = m$.

Case (*BCD var*)**:** The expression m is a variable and the result holds with $e = m$.

Case (*BCD sub*)**:** We have

$$D = \frac{D' \qquad t' \leq_{BCD} t}{\Gamma \vdash_{BCD} m : t}$$

where D' in intersection-abstraction normal form and proves $\Gamma \vdash_{BCD} m : t'$. Clearly we have $|D'| < |D|$ and $\mathsf{var}(D') \subseteq \Delta$, so by the induction hypothesis, there exists e such that $\lceil e \rceil = m$ and $\Delta \fatsemi \Gamma \vdash e : t'$. By Lemma B.18, we have $t' \leq t$, therefore we have $\Delta \fatsemi \Gamma \vdash e : t$, as wished.

Case (*BCD app*)**:** We have

$$D = \frac{D_1 \qquad D_2}{\Gamma \vdash_{BCD} m : t}$$

where D_1 proves $\Gamma \vdash_{BCD} m_1 : s \to t$, D_2 proves $\Gamma \vdash_{BCD} m_2 : s$, $m = m_1\ m_2$, and both D_1 and D_2 are in intersection-abstraction normal form. Since $|D_i| < |D|$ and $\mathsf{var}(D_i) \subseteq \Delta$, by the induction hypothesis, there exist e_1 and e_2 such that $\lceil e_1 \rceil = m_1$, $\lceil e_2 \rceil = m_2$, $\Delta \fatsemi \Gamma \vdash e_1 : s \to t$, and $\Delta \fatsemi \Gamma \vdash e_2 : s$. Consequently we have $\Delta \fatsemi \Gamma \vdash e_1\ e_2 : t$, with $\lceil e_1\ e_2 \rceil = m_1\ m_2$, as wished.

Case (*BCD abstr*) **(or** (*BCD inter*)**):** Because D is in intersection-abstraction normal form, we have

$$D = \frac{\dfrac{D_i}{\Gamma \vdash_{BCD} \lambda x.m' : s_i \to t_i}\, i \in I}{\Gamma \vdash_{BCD} m : t}$$

where each D_i is in intersection-abstraction normal form and proves $\Gamma, x : s_i \vdash_{BCD} m' : t_i$, $t = \bigwedge_{i\in I} s_i \to t_i$, and $m = \lambda x.m'$. Since $\bigcup_{i\in I} \mathsf{var}(D_i) \subseteq \Delta$, by Lemma B.22, there exist e', $\sigma_1, \ldots, \sigma_n$ such that $\lceil e' \rceil = m'$, $\mathsf{dom}(\sigma_1) = \ldots = \mathsf{dom}(\sigma_n) \subseteq \mathsf{tv}(e')$, and $\Delta \fatsemi \Gamma, x : s_i \vdash e'@[\sigma_i] : t_i$ for all $i \in I$. Let α, β be two fresh type variables. We define $\sigma'_i = \sigma_i \cup \{s_i/\alpha, t_i/\beta\}$ for all $i \in I$. Because $\mathsf{dom}(\sigma_i) \cap \{\alpha, \beta\} = \emptyset$ and $\mathsf{tv}(e') \cap \{\alpha, \beta\} = \emptyset$, by Lemma B.9 we have $\Delta \fatsemi \Gamma, x : s_i \vdash e'@[\sigma'_i] : t_i$, which is equivalent to $\Delta \fatsemi \Gamma, x : \alpha\sigma'_i \vdash e'@[\sigma'_i] : \beta\sigma'_i$. Note that $\Delta \cup \mathsf{var}(\bigwedge_{i\in I}(\alpha \to \beta)\sigma'_i) = \Delta \cup \mathsf{var}(\bigwedge_{i\in I} s_i \to t_i) = \Delta$ by definition of Δ, so by rule *(abstr)*, we have $\Delta \fatsemi \Gamma \vdash \lambda^{\alpha\to\beta}_{[\sigma'_i]_{i\in I}} x.e' : \bigwedge_{i\in I} s_i \to t_i$. Hence we have the required result with $e = \lambda^{\alpha\to\beta}_{[\sigma'_i]_{i\in I}} x.e'$.

□

B.4 Elimination of sets of type-substitutions

In this section we prove that the expressions of the form $e[\sigma_j]_{j\in J}$ are redundant insofar as their presence in the calculus does not increase its expressive power. For that we consider a subcalculus, called *normalized calculus*, in which sets of type-substitutions appear only in decorations.

Definition B.24. *A* normalized expression e *is an expression without any subterm of the form* $e[\sigma_j]_{j\in J}$*, i.e., an expression respecting the following grammar:*

$$e ::= c \mid x \mid (e, e) \mid \pi_i(e) \mid e\, e \mid \lambda^{\wedge_{i\in I} s_i \to t_i}_{[\sigma_j]_{j\in J}} x.e \mid e \in t \,?\, e : e$$

The set of all normalized expressions is denoted as $\mathscr{E}_N$.

We then define an embedding of the full calculus into this subcalculus as follows:

Definition B.25. *The* embedding *emd*$(.)$ *is mapping from* $\mathscr{E}$ *to* $\mathscr{E}_N$ *defined as*

$$\begin{aligned}
\mathit{emd}(c) &= c \\
\mathit{emd}(x) &= x \\
\mathit{emd}((e_1, e_2)) &= (\mathit{emd}(e_1), \mathit{emd}(e_2)) \\
\mathit{emd}(\pi_i(e)) &= \pi_i(\mathit{emd}(e)) \\
\mathit{emd}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) &= \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.\mathit{emd}(e) \\
\mathit{emd}(e_1 e_2) &= \mathit{emd}(e_1)\mathit{emd}(e_2) \\
\mathit{emd}(e \in t \,?\, e_1 : e_2) &= \mathit{emd}(e) \in t \,?\, \mathit{emd}(e_1) : \mathit{emd}(e_2) \\
\mathit{emd}(e[\sigma_j]_{j\in J}) &= \mathit{emd}(e@[\sigma_j]_{j\in J})
\end{aligned}$$

We want to prove that the subcalculus has the same expressive power as the full calculus, namely, given an expression and its embedding, they reduce to the same value. We proceed in several steps, using auxiliary lemmas.

First we show that the embedding preserves values.

Lemma B.26. *Let* $v \in \mathscr{V}$ *be a value. Then* $\mathit{emd}(v) \in \mathscr{V}$.

Proof. Straightforward. □

Then we prove that values and their embeddings have the same types.

Lemma B.27. *Let* $v \in \mathscr{V}$ *be a value. Then* $\vdash v : t \iff \vdash \mathit{emd}(v) : t$.

Proof. By induction and case analysis on v (note that *emd*$(.)$ does not change the types in the interfaces). □

We now want to prove the embedding preserves reduction, that is if an expression e reduces to e' in the full calculus, then its embedding *emd*(e) reduces to *emd*(e') in the subcalculus. Before that we show a substitution lemma.

Lemma B.28. *Let* e *be an expression,* x *an expression variable and* v *a value. Then* $\mathit{emd}(e\{v/x\}) = \mathit{emd}(e)\{\mathit{emd}(v)/x\}$.

Proof. By induction and case analysis on e.

$\underline{c}$**:**

$$\begin{aligned}
\mathit{emd}(c\{v/x\}) &= \mathit{emd}(c) \\
&= c \\
&= c\{\mathit{emd}(v)/x\} \\
&= \mathit{emd}(c)\{\mathit{emd}(v)/x\}
\end{aligned}$$

$\underline{y}$**:**

$$\begin{aligned} \mathit{emd}(y\{v/x\}) &= \mathit{emd}(y) \\ &= y \\ &= y\{\mathit{emd}(v)/x\} \\ &= \mathit{emd}(y)\{\mathit{emd}(v)/x\} \end{aligned}$$

$\underline{x}$**:**

$$\begin{aligned} \mathit{emd}(x\{v/x\}) &= \mathit{emd}(v) \\ &= x\{\mathit{emd}(v)/x\} \\ &= \mathit{emd}(x)\{\mathit{emd}(v)/x\} \end{aligned}$$

$\underline{(e_1, e_2)}$**:**

$$\begin{aligned} \mathit{emd}((e_1,e_2)\{v/x\}) &= \mathit{emd}((e_1\{v/x\}, e_2\{v/x\})) \\ &= (\mathit{emd}(e_1\{v/x\}), \mathit{emd}(e_2\{v/x\})) \\ &= (\mathit{emd}(e_1)\{\mathit{emd}(v)/x\}, \mathit{emd}(e_2)\{\mathit{emd}(v)/x\}) \quad \text{(by induction)} \\ &= (\mathit{emd}(e_1), \mathit{emd}(e_2))\{\mathit{emd}(v)/x\} \\ &= \mathit{emd}((e_1,e_2))\{\mathit{emd}(v)/x\} \end{aligned}$$

$\underline{\pi_i(e')}$**:**

$$\begin{aligned} \mathit{emd}(\pi_i(e')\{v/x\}) &= \mathit{emd}(\pi_i(e'\{v/x\})) \\ &= \pi_i(\mathit{emd}(e'\{v/x\})) \\ &= \pi_i(\mathit{emd}(e')\{\mathit{emd}(v)/x\}) \quad \text{(by induction)} \\ &= \pi_i(\mathit{emd}(e'))\{\mathit{emd}(v)/x\} \\ &= \mathit{emd}(\pi_i(e'))\{\mathit{emd}(v)/x\} \end{aligned}$$

$\underline{e_1 e_2}$**:**

$$\begin{aligned} \mathit{emd}((e_1e_2)\{v/x\}) &= \mathit{emd}((e_1\{v/x\})(e_2\{v/x\})) \\ &= \mathit{emd}(e_1\{v/x\})\mathit{emd}(e_2\{v/x\}) \\ &= (\mathit{emd}(e_1)\{\mathit{emd}(v)/x\})(\mathit{emd}(e_2)\{\mathit{emd}(v)/x\}) \quad \text{(by induction)} \\ &= (\mathit{emd}(e_1)\mathit{emd}(e_2))\{\mathit{emd}(v)/x\} \\ &= \mathit{emd}(e_1e_2)\{\mathit{emd}(v)/x\} \end{aligned}$$

$\underline{\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} y.e_0}$**:** using α-conversion, we can assume that $\mathsf{tv}(v) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$.

$$\begin{aligned} \mathit{emd}((\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} y.e_0)\{v/x\}) &= \mathit{emd}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} y.e_0\{v/x\}) \\ &= \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} y.\mathit{emd}(e_0\{v/x\}) \\ &= \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} y.(\mathit{emd}(e_0)\{\mathit{emd}(v)/x\}) \quad \text{(by induction)} \\ &= (\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} y.\mathit{emd}(e_0))\{\mathit{emd}(v)/x\} \\ &= (\mathit{emd}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} y.e_0))\{\mathit{emd}(v)/x\} \end{aligned}$$

$\underline{e_0 \in t\,?\,e_1 : e_2}$**:**

$$\begin{aligned} & \mathit{emd}((e_0 \in t\,?\,e_1 : e_2)\{v/x\}) \\ = \;& \mathit{emd}((e_0\{v/x\}) \in t\,?\,(e_1\{v/x\}) : (e_2\{v/x\})) \\ = \;& \mathit{emd}(e_0\{v/x\}) \in t\,?\,\mathit{emd}(e_1\{v/x\}) : \mathit{emd}(e_2\{v/x\}) \\ = \;& \mathit{emd}(e_0)\{\mathit{emd}(v)/x\} \in t\,?\,(\mathit{emd}(e_1)\{\mathit{emd}(v)/x\}) : (\mathit{emd}(e_2)\{\mathit{emd}(v)/x\}) \quad \text{(by induction)} \\ = \;& (\mathit{emd}(e_0) \in t\,?\,\mathit{emd}(e_1) : \mathit{emd}(e_2))\{\mathit{emd}(v)/x\} \\ = \;& \mathit{emd}(e_0 \in t\,?\,e_1 : e_2)\{\mathit{emd}(v)/x\} \end{aligned}$$

$\underline{e'[\sigma_j]_{j\in J}}$**:** using α-conversion, we can assume that $\mathsf{tv}(v) \cap \bigcup_{j\in J} \mathsf{dom}(\sigma_j) = \emptyset$

$$\begin{aligned} \mathit{emd}((e'[\sigma_j]_{j\in J})\{v/x\}) &= \mathit{emd}((e'\{v/x\})[\sigma_j]_{j\in J}) \\ &= \mathit{emd}((e'\{v/x\})@[\sigma_j]_{j\in J}) \\ &= \mathit{emd}((e'@[\sigma_j]_{j\in J})\{v/x\}) \quad \text{(Lemma B.5)} \\ &= \mathit{emd}(e'@[\sigma_j]_{j\in J})\{\mathit{emd}(v)/x\} \quad \text{(by induction)} \\ &= \mathit{emd}(e'[\sigma_j]_{j\in J})\{\mathit{emd}(v)/x\} \end{aligned}$$

□

Lemma B.29. *Let $e \in \mathscr{E}$ be an expression. If $e \rightsquigarrow e'$, then $\mathit{emd}(e) \rightsquigarrow^* \mathit{emd}(e')$.*

Proof. By induction and case analysis on e.

$\underline{c}$**,** $\underline{x}$**:** irreducible.

$\underline{(e_1, e_2)}$**:** there are two ways to reduce e:

(1) $e_1 \rightsquigarrow e_1'$. By induction, $\mathit{emd}(e_1) \rightsquigarrow^* \mathit{emd}(e_1')$. Then we have $(\mathit{emd}(e_1), \mathit{emd}(e_2)) \rightsquigarrow^* (\mathit{emd}(e_1'), \mathit{emd}(e_2))$, that is, $\mathit{emd}((e_1, e_2)) \rightsquigarrow^* \mathit{emd}((e_1', e_2))$.

(2) $e_2 \rightsquigarrow e_2'$. Similar to the subcase above.

$\underline{\pi_i(e_0)}$**:** there are two ways to reduce e:

(1) $e_0 \rightsquigarrow e_0'$. By induction, $\mathit{emd}(e_0) \rightsquigarrow^* \mathit{emd}(e_0')$. Then we have $\pi_i(\mathit{emd}(e_0)) \rightsquigarrow^* \pi_i(\mathit{emd}(e_0'))$, that is, $\mathit{emd}(\pi_i(e_0)) \rightsquigarrow^* \mathit{emd}(\pi_i(e_0'))$.

(2) $e_0 = (v_1, v_2)$ and $e \rightsquigarrow v_i$. According to Lemma B.26, $\mathit{emd}((v_1, v_2)) \in \mathscr{V}$. Moreover, $\mathit{emd}((v_1, v_2)) = (\mathit{emd}(v_1), \mathit{emd}(v_2))$. Therefore, $\pi_i(\mathit{emd}(v_1), \mathit{emd}(v_2)) \rightsquigarrow \mathit{emd}(v_i)$, which is the same as $\mathit{emd}(\pi_i(v_1, v_2)) \rightsquigarrow \mathit{emd}(v_i)$.

$\underline{e_1 e_2}$**:** there are three ways to reduce e:

(1) $e_1 \rightsquigarrow e_1'$. Similar to the case of (e_1, e_2).

(2) $e_2 \rightsquigarrow e_2'$. Similar to the case of (e_1, e_2).

(3) $e_1 = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$, $e_2 = v_2$ and $e_1 e_2 \rightsquigarrow (e_0@[\sigma_j]_{j\in P})\{v_2/x\}$, where $P = \{j \in J \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_j\}$. According to Lemma B.27, we have $\vdash v_2 : t_i\sigma_j \iff \vdash \mathit{emd}(v_2) : t_i\sigma_j$, thus we have

$$\{j \in J \mid \exists i \in I.\ \vdash \mathit{emd}(v_2) : t_i\sigma_j\} = \{j \in J \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_j\}.$$

Therefore, $\mathit{emd}(e_1)\mathit{emd}(v_2) \rightsquigarrow \mathit{emd}(e_0@[\sigma_j]_{j\in P})\{\mathit{emd}(v_2)/x\}$. Moreover, by lemma B.28, we can get

$$\mathit{emd}(e_0@[\sigma_j]_{j\in P})\{\mathit{emd}(v_2)/x\} = \mathit{emd}(e_0@[\sigma_j]_{j\in P}\{v_2/x\}),$$

which proves this case.

$\underline{\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0}$**:** It cannot be reduced. Thus the result follows.

$\underline{e_0 \in t\,?\,e_1 : e_2}$**:** there are three ways to reduce e:

(1) $e_i \rightsquigarrow e_i'$. Similar to the case of (e_1, e_2).

(2) $e_0 = v_0$, $\vdash v_0 : t$ and $e \rightsquigarrow e_1$. According to Lemmas B.26 and B.27, $\mathit{emd}(v_0) \in \mathscr{V}$ and $\vdash \mathit{emd}(v_0) : t$. So we have $\mathit{emd}(v_0) \in t\,?\,\mathit{emd}(e_1) : \mathit{emd}(e_2) \rightsquigarrow \mathit{emd}(e_1)$.

(3) $e_0 = v_0$, $\nvdash v_0 : t$ and $e \rightsquigarrow e_2$. According to Lemmas B.26 and B.27, $\mathit{emd}(v_0) \in \mathscr{V}$ and $\nvdash \mathit{emd}(v_0) : t$. Therefore, $\mathit{emd}(v_0) \in t\,?\,\mathit{emd}(e_1) : \mathit{emd}(e_2) \rightsquigarrow \mathit{emd}(e_2)$.

$\underline{e_0[\sigma_j]_{j\in J}}$**:** $e \rightsquigarrow e_0@[\sigma_j]_{j\in J}$. By Definition B.25, $\mathit{emd}(e_0[\sigma_j]_{j\in J}) = \mathit{emd}(e_0@[\sigma_j]_{j\in J})$. Therefore, the result follows.

□

Although the embedding preserves the reduction, it does not indicate that an expression and its embedding reduce to the same value. This is because that there may be some subterms of the form $e[\sigma_j]_{j\in J}$ in the body expression of an abstraction value. For example, the expression

$$(\lambda^{\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}} z.\lambda^{\mathtt{Int}\to\mathtt{Int}} y.((\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/\alpha\}]42))3$$

reduces to

$$\lambda^{\mathtt{Int}\to\mathtt{Int}} y.((\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/\alpha\}]42),$$

while its embedding reduces to

$$\lambda^{\mathtt{Int}\to\mathtt{Int}} y.((\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} x.x)42).$$

However, the embedding of the value returned by an expression is the value returned by the embedding of the expression. For instance, consider the example above again:

$$\mathit{emd}(\lambda^{\mathtt{Int}\to\mathtt{Int}} y.((\lambda^{\alpha\to\alpha} x.x)[\{\mathtt{Int}/\alpha\}]42)) = \lambda^{\mathtt{Int}\to\mathtt{Int}} y.((\lambda^{\alpha\to\alpha}_{[\{\mathtt{Int}/\alpha\}]} x.x)42)$$

Next, we want to prove an inversion of Lemma B.29, that is, if the embedding $\mathit{emd}(e)$ of an expression e reduces to e', then there exists e'' such that its embedding is e' and e reduces to e''. Prior to that we prove two auxiliary lemmas: the inversions for values and for relabeled expressions.

Lemma B.30. *Let $e \in \mathscr{E}$ an expression. If $\mathit{emd}(e) \in \mathscr{V}$, then there exists a value $v \in \mathscr{V}$ such that $e \rightsquigarrow^*_{(\mathit{Rinst})} v$ and $\mathit{emd}(e) = \mathit{emd}(v)$. More specifically,*

(1) *if $\mathit{emd}(e) = c$, then $e \rightsquigarrow^*_{(\mathit{Rinst})} c$ and $\mathit{emd}(e) = c$.*

(2) *if $\mathit{emd}(e) = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$, then there exists e_0' such that $e \rightsquigarrow^*_{(\mathit{Rinst})} \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0'$ and $\mathit{emd}(e_0') = e_0$.*

(3) *if $\mathit{emd}(e) = (v_1, v_2)$, then there exist v_1, v_2 such that $e \rightsquigarrow^*_{(\mathit{Rinst})} (v_1', v_2')$ and $\mathit{emd}(v_i') = v_i$.*

Proof. By induction and case analysis on $\mathit{emd}(e)$.

$\underline{c}$**:** according to Definition B.25, e should be the form of $c[\sigma_{j_1}]_{j_1\in J_1} \ldots [\sigma_{j_n}]_{j_n\in J_n}$, where $n \geq 0$. Clearly, we have $e \rightsquigarrow^*_{(\mathit{Rinst})} c$ and $\mathit{emd}(e) = c$.

$\underline{\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0}$**:** according to Definition B.25, e should be the form of

$$(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_{j_0}]_{j_0\in J_0}} x.e_0')[\sigma_{j_1}]_{j_1\in J_1} \ldots [\sigma_{j_n}]_{j_n\in J_n}$$

where $\mathit{emd}(e_0') = e_0$, $[\sigma_{j_n}]_{j_n\in J_n} \circ \ldots \circ [\sigma_{j_1}]_{j_1\in J_1} \circ [\sigma_{j_0}]_{j_0\in J_0} = [\sigma_j]_{j\in J}$, and $n \geq 0$. Moreover, it is clear that $e \rightsquigarrow^*_{(\mathit{Rinst})} \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0'$. Let $v = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0'$ and the result follows.

$\underline{(v_1, v_2)}$**:** according to Definition B.25, e should be the form of $(e_1, e_2)[\sigma_{j_1}]_{j_1\in J_1}\dots[\sigma_{j_n}]_{j_n\in J_n}$, where $\mathit{emd}(e_i@[\sigma_{j_1}]_{j_1\in J_1}@\dots@[\sigma_{j_n}]_{j_n\in J_n}) = v_i$ and $n \geq 0$. Moreover, it is easy to get that

$$e \rightsquigarrow^*_{(\mathit{Rinst})} (e_1@[\sigma_{j_1}]_{j_1\in J_1}@\dots@[\sigma_{j_n}]_{j_n\in J_n}, e_2@[\sigma_{j_1}]_{j_1\in J_1}@\dots@[\sigma_{j_n}]_{j_n\in J_n})$$

By induction on v_i, there exists v_i' such that $e_i@[\sigma_{j_1}]_{j_1\in J_1}@\dots@[\sigma_{j_n}]_{j_n\in J_n} \rightsquigarrow^*_{(\mathit{Rinst})} v_i'$ and $\mathit{emd}(e_i@[\sigma_{j_1}]_{j_1\in J_1}@\dots@[\sigma_{j_n}]_{j_n\in J_n}) = \mathit{emd}(v_i')$. Let $v = (v_1', v_2')$. Then we have $e \rightsquigarrow^*_{(\mathit{Rinst})} (v_1', v_2')$ and $\mathit{emd}(v) = (\mathit{emd}(v_1'), \mathit{emd}(v_2')) = (v_1, v_2) = \mathit{emd}(e)$. Therefore, the result follows.

□

Lemma B.31. *Let $e \in \mathscr{E}$ be an expression and $[\sigma_j]_{j\in J}$ a set of substitutions. If $\mathit{emd}(e@[\sigma_j]_{j\in J}) \rightsquigarrow e'$, then there exists e'' such that $e@[\sigma_j]_{j\in J} \rightsquigarrow^+ e''$ and $\mathit{emd}(e'') = e'$.*

Proof. By induction and case analysis on e.

$\underline{c, x}$**:** straightforward.

$\underline{(e_1, e_2)}$**:** $\mathit{emd}(e@[\sigma_j]_{j\in J}) = (\mathit{emd}(e_1@[\sigma_j]_{j\in J}), \mathit{emd}(e_2@[\sigma_j]_{j\in J}))$. There are two ways to reduce $\mathit{emd}(e@[\sigma_j]_{j\in J})$:

(1) $\mathit{emd}(e_1@[\sigma_j]_{j\in J}) \rightsquigarrow e_1'$. By induction, there exists e_1'' such that $e_1@[\sigma_j]_{j\in J} \rightsquigarrow^+ e_1''$ and $\mathit{emd}(e_1'') = e_1'$. Then we have $(e_1@[\sigma_j]_{j\in J}, e_2@[\sigma_j]_{j\in J}) \rightsquigarrow^+ (e_1'', e_2@[\sigma_j]_{j\in J})$ and $\mathit{emd}((e_1'', e_2@[\sigma_j]_{j\in J})) = (e_1', \mathit{emd}(e_2@[\sigma_j]_{j\in J}))$.

(2) $\mathit{emd}(e_2@[\sigma_j]_{j\in J}) \rightsquigarrow e_2'$. Similar to the subcase above.

$\underline{\pi_i(e_0)}$**:** $\mathit{emd}(e@[\sigma_j]_{j\in J}) = \pi_i(\mathit{emd}(e_0@[\sigma_j]_{j\in J}))$. There are two ways to reduce $\mathit{emd}(e@[\sigma_j]_{j\in J})$:

(1) $\mathit{emd}(e_0@[\sigma_j]_{j\in J}) \rightsquigarrow e_0'$. By induction, there exists e_0'' such that $e_0@[\sigma_j]_{j\in J} \rightsquigarrow^+ e_0''$ and $\mathit{emd}(e_0'') = e_0'$. Then we have $\pi_i(e_0@[\sigma_j]_{j\in J}) \rightsquigarrow^+ \pi_i(e_0'')$ and $\mathit{emd}(\pi_i(e_0'')) = \pi_i(e_0')$.

(2) $\mathit{emd}(e_0@[\sigma_j]_{j\in J}) = (v_1, v_2)$ and $\mathit{emd}(e@[\sigma_j]_{j\in J}) \rightsquigarrow v_i$. According to Lemma B.30, there exist v_1' and v_2' such that $e_0@[\sigma_j]_{j\in J} \rightsquigarrow^*_{(\mathit{Rinst})} (v_1', v_2')$ and $\mathit{emd}(v_i') = v_i$. Then $\pi_i(e_0@[\sigma_j]_{j\in J}) \rightsquigarrow^+ v_i'$. The result follows.

$\underline{e_1e_2}$**:** $\mathit{emd}(e@[\sigma_j]_{j\in J}) = \mathit{emd}(e_1@[\sigma_j]_{j\in J})\mathit{emd}(e_2@[\sigma_j]_{j\in J})$. There are three possible ways to reduce $\mathit{emd}(e@[\sigma_j]_{j\in J})$:

(1) $\mathit{emd}(e_1@[\sigma_j]_{j\in J}) \rightsquigarrow e_1'$. Similar to the case of (e_1, e_2).

(2) $\mathit{emd}(e_2@[\sigma_j]_{j\in J}) \rightsquigarrow e_2'$. Similar to the case of (e_1, e_2).

(3) $\mathit{emd}(e_1@[\sigma_j]_{j\in J}) = \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e_0$, $\mathit{emd}(e_2@[\sigma_j]_{j\in J}) = v_2$ and

$$\mathit{emd}(e@[\sigma_j]_{j\in J}) \rightsquigarrow (e_0@[\sigma_k]_{k\in P})\{v_2/x\},$$

where $P = \{k \in K \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_k\}$. According to Lemma B.30, we have (i) there exists e_0' such that $e_1@[\sigma_j]_{j\in J} \rightsquigarrow^*_{(\mathit{Rinst})} \lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e_0'$ and $\mathit{emd}(e_0') = e_0$; and (ii) there exists v_2' such that $e_2@[\sigma_j]_{j\in J} \rightsquigarrow^*_{(\mathit{Rinst})} v_2'$ and $\mathit{emd}(v_2') = v_2$. Moreover, by Lemma B.27, we get $\vdash v_2 : t_i\sigma_k \iff \vdash v_2' : t_i\sigma_k$, thus $\{k \in K \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_k\} = \{k \in K \mid \exists i \in I.\ \vdash v_2' : t_i\sigma_k\}$. Therefore, $e@[\sigma_j]_{j\in J} \rightsquigarrow^+ (e_0'@[\sigma_k]_{k\in P})\{v_2'/x\}$. Finally, by lemma B.28, $\mathit{emd}(e_0'@[\sigma_k]_{k\in P}\{v_2'/x\}) = \mathit{emd}(e_0')@[\sigma_k]_{k\in P}\{\mathit{emd}(v_2')/x\} = e_0@[\sigma_k]_{k\in P}\{v_2/x\}$.

$\underline{\lambda^{\wedge_{i\in I} t_i\to s_i}_{[\sigma_k]_{k\in K}} x.e_0}$**:** It cannot be reduced. Thus the result follows.

$\underline{e_0 \in t\ ?\ e_1 : e_2}$**:** $\mathit{emd}(e@[\sigma_j]_{j\in J}) = \mathit{emd}(e_0@[\sigma_j]_{j\in J}) \in t\ ?\ \mathit{emd}(e_1@[\sigma_j]_{j\in J}) : \mathit{emd}(e_2@[\sigma_j]_{j\in J})$. There are three ways to reduce $\mathit{emd}(e@[\sigma_j]_{j\in J})$:

(1) $\mathit{emd}(e_i@[\sigma_j]_{j\in J}) \rightsquigarrow e_i'$. Similar to the case of (e_1, e_2).

(2) $\mathit{emd}(e_0@[\sigma_j]_{j\in J}) = v_0, \vdash v_0 : t$ and $\mathit{emd}(e@[\sigma_j]_{j\in J}) \rightsquigarrow \mathit{emd}(e_1@[\sigma_j]_{j\in J})$. According to Lemma B.30, there exists v_0' such that $e_0@[\sigma_j]_{j\in J} \rightsquigarrow^*_{(\mathit{Rinst})} v_0'$ and $\mathit{emd}(v_0') = v_0$. Moreover, by Lemma B.27, $\vdash v_0' : t$. So $e@[\sigma_j]_{j\in J} \rightsquigarrow e_1@[\sigma_j]_{j\in J}$.

(3) $\mathit{emd}(e_0@[\sigma_j]_{j\in J}) = v_0, \nvdash v_0 : t$ and $\mathit{emd}(e@[\sigma_j]_{j\in J}) \rightsquigarrow \mathit{emd}(e_2@[\sigma_j]_{j\in J})$. Similar to the subcase above.

$\underline{e_0[\sigma_k]_{k\in K}}$**:** $\mathit{emd}(e@[\sigma_j]_{j\in J}) = \mathit{emd}(e_0@([\sigma_j]_{j\in J} \circ [\sigma_k]_{k\in K}))$. By induction, the result follows.

□

Lemma B.32. *Let $e \in \mathscr{E}$ be an expression. If $\mathit{emd}(e) \rightsquigarrow e'$, then there exists e'' such that $e \rightsquigarrow^+ e''$ and $\mathit{emd}(e'') = e'$.*

Proof. By induction and case analysis on e.

$\underline{c, x}$**:** straightforward.

$\underline{(e_1, e_2)}$**:** $\mathit{emd}(e) = (\mathit{emd}(e_1), \mathit{emd}(e_2))$. There are two ways to reduce $\mathit{emd}(e)$:

(1) $\mathit{emd}(e_1) \rightsquigarrow e_1'$. By induction, there exists e_1'' such that $e_1 \rightsquigarrow^+ e_1''$ and $\mathit{emd}(e_1'') = e_1'$. Then we have $(e_1, e_2) \rightsquigarrow^+ (e_1'', e_2)$ and $\mathit{emd}((e_1'', e_2)) = (e_1', \mathit{emd}(e_2))$.

(2) $\mathit{emd}(e_2) \rightsquigarrow e_2'$. Similar to the subcase above.

$\underline{\pi_i(e_0)}$**:** $emd(e) = \pi_i(emd(e_0))$. There are two ways to reduce $emd(e)$:
(1) $emd(e_0) \rightsquigarrow e_0'$. By induction, there exists e_0'' such that $e_0 \rightsquigarrow^+ e_0''$ and $emd(e_0'') = e_0'$. Then we have $\pi_i(e_0) \rightsquigarrow^+ \pi_i(e_0'')$ and $emd(\pi_i(e_0'')) = \pi_i(e_0')$.
(2) $emd(e_0) = (v_1, v_2)$ and $emd(e) \rightsquigarrow v_i$. According to Lemma B.30, there exist v_1' and v_2' such that $e_0 \rightsquigarrow^*_{(Rinst)} (v_1', v_2')$ and $emd(v_i') = v_i$. Then $\pi_i(e_0) \rightsquigarrow^+ v_i'$. The result follows.

$\underline{e_1 e_2}$**:** $emd(e) = emd(e_1)emd(e_2)$. There are three ways to reduce $emd(e)$:
(1) $emd(e_1) \rightsquigarrow e_1'$. Similar to the case of (e_1, e_2).
(2) $emd(e_2) \rightsquigarrow e_2'$. Similar to the case of (e_1, e_2).
(3) $emd(e_1) = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0$, $emd(e_2) = v_2$ and $emd(e) \rightsquigarrow (e_0@[\sigma_j]_{j\in P})\{v_2/x\}$, where $P = \{j \in J \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_j\}$. According to Lemma B.30, we have (i) there exists e_0' such that $e_1 \rightsquigarrow^*_{(Rinst)} \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_k]_{k\in K}} x.e_0'$ and $emd(e_0') = e_0$; and (ii) there exists v_2' such that $e_2 \rightsquigarrow^*_{(Rinst)} v_2'$ and $emd(v_2') = v_2$. Moreover, by Lemma B.27, we get $\vdash v_2 : t_i\sigma_j \iff \vdash v_2' : t_i\sigma_j$, thus $\{j \in J \mid \exists i \in I.\ \vdash v_2 : t_i\sigma_j\} = \{j \in J \mid \exists i \in I.\ \vdash v_2' : t_i\sigma_j\}$. Therefore, $e \rightsquigarrow^+ (e_0'@[\sigma_j]_{j\in P})\{v_2'/x\}$. Finally, by lemma B.28, $emd(e_0'@[\sigma_j]_{j\in P}\{v_2'/x\}) = emd(e_0')@[\sigma_j]_{j\in P}\{emd(v_2')/x\} = e_0@[\sigma_j]_{j\in P}\{v_2/x\}$.

$\underline{\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e_0}$**:** It cannot be reduced. Thus the result follows.

$\underline{e_0 \in t\ ?\ e_1 : e_2}$**:** $emd(e) = emd(e_0) \in t\ ?\ emd(e_1) : emd(e_2)$. There are three ways to reduce $emd(e)$:
(1) $emd(e_i) \rightsquigarrow e_i'$. Similar to the case of (e_1, e_2).
(2) $emd(e_0) = v_0, \vdash v_0 : t$ and $emd(e) \rightsquigarrow emd(e_1)$. According to Lemma B.30, there exists v_0' such that $e_0 \rightsquigarrow^*_{(Rinst)} v_0'$ and $emd(v_0') = v_0$. Moreover, by Lemma B.27, $\vdash v_0' : t$. So $e \rightsquigarrow e_1$.
(3) $emd(e_0) = v_0, \nvdash v_0 : t$ and $emd(e) \rightsquigarrow emd(e_2)$. Similar to the subcase above.

$\underline{e_0[\sigma_j]_{j\in J}}$**:** $emd(e_0[\sigma_j]_{j\in J}) = emd(e_0@[\sigma_j]_{j\in J})$ and $e_0[\sigma_j]_{j\in J} \rightsquigarrow e_0@[\sigma_j]_{j\in J}$. By Lemma B.31, the result follows.

□

Thus we have the following theorem

Theorem B.33. *Let $e \in \mathscr{E}$ be an expression.*

(1) *if $e \rightsquigarrow^* v$, then $emd(e) \rightsquigarrow^* emd(v)$.*
(2) *if $emd(e) \rightsquigarrow^* v$, then there exists $v' \in \mathscr{V}$ such that $e \rightsquigarrow^* v'$ and $emd(v') = v$.*

Proof. (1)**:** By induction on the reduction and by Lemma B.29.
(2)**:** By induction on the reduction and by Lemma B.32.

□

In addition, it is easy to prove that the subcalculus $\mathscr{E}_N$ is closed under the reduction rules, and we can safely disregard (*Rinst*) since it cannot be applied. Then the normalized calculus also possess, for example, the soundness property.

C. Algorithmic Type Checking

The typing rules provided in Section A.3 are not syntax-directed because of the presence of the subsumption rule. In this section we present an equivalent type system with syntax-directed rules. In order to define it we consider the rules of Section A.3. First, we merge the rules (*inst*) and (*inter*) into one rule (since we prove that intersection is interesting only to merge different instances of a same type), and then we consider where subsumption is used and whether it can be postponed by moving it down the derivation tree.

C.1 Merging Intersection and Instantiation

Intersection is used to merge different types derived for the same term. In this calculus, we can derive different types for a term because of either subsumption or instantiation. However, the intersection of different super-types can be obtained by subsumption itself (if $t \leq t_1$ and $t \leq t_2$, then $t \leq t_1 \wedge t_2$), so intersection is really useful only to merge different instances of a same type, as we can see with rule (*inter*) in Figure 3. Note that all the subjects in the premise of (*inter*) share the same structure $e[\sigma]$, and the typing derivations of these terms must end with either (*inst*) or (*subsum*). We show that we can in fact postpone the uses of (*subsum*) after (*inter*), and we can therefore merge the rules (*inst*) and (*inter*) into one rule (*instinter*) as follows:

$$\frac{\Delta \fatsemi \Gamma \vdash e : t \qquad \forall j \in J.\ \sigma_j \sharp \Delta \qquad |J| > 0}{\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j} \ (\textit{instinter})$$

Let $\Delta \fatsemi \Gamma \vdash_m e : t$ denote the typing judgments derivable in the type system with the typing rule (*instinter*) but not (*inst*) and (*inter*). The following theorem proves that the type system $\vdash_m$ (m stands for "merged") is equivalent to the original one $\vdash$.

Theorem C.1. *Let e be an expression. Then $\Delta \fatsemi \Gamma \vdash_m e : t \iff \Delta \fatsemi \Gamma \vdash e : t$.*

Proof. $\Rightarrow$**:** It is clear that (*inst*) is a special case of (*instinter*) where $|J| = 1$. We simulate each instance of (*instinter*) where $|J| > 1$ by using several instances of (*inst*) followed by one instance of (*inter*). In detail, consider the following derivation

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta' \wr \Gamma' \vdash e' : t'} \quad \sigma_j \sharp \Delta'}{\Delta' \wr \Gamma' \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j}\ (instinter)}{\dfrac{\dots \qquad \vdots \qquad \dots}{\Delta \wr \Gamma \vdash e : t}}$$

We can rewrite this derivation as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Delta' \wr \Gamma' \vdash e' : t'} \quad \sigma_1 \sharp \Delta'}{\Delta' \wr \Gamma' \vdash e'[\sigma_1] : t'\sigma_1}\ (inst) \quad \dots \quad \dfrac{\dfrac{\dots}{\Delta' \wr \Gamma' \vdash e' : t'} \quad \sigma_{|J|} \sharp \Delta'}{\Delta' \wr \Gamma' \vdash e'[\sigma_{|J|}] : t'\sigma_{|J|}}\ (inst)}{\Delta' \wr \Gamma' \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t'\sigma_j}\ (inter)}{\dfrac{\dots \qquad \vdots \qquad \dots}{\Delta \wr \Gamma \vdash e : t}}$$

$\Leftarrow$**:** The proof proceeds by induction and case analysis on the structure of e. For each case we use an auxiliary internal induction on the typing derivation. We label **E** the main (external) induction and **I** the internal induction in what follows.

$\underline{e = c}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*const*) or (*subsum*). If the typing derivation ends with (*const*), the result follows straightforward.
Otherwise, the typing derivation ends with an instance of (*subsum*):

$$\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e : s} \quad s \leq t}{\Delta \wr \Gamma \vdash e : t}\ (subsum)$$

Then by **I**-induction, we have $\Delta \wr \Gamma \vdash_m e : s$. Since $s \leq t$, by subsumption, we get $\Delta \wr \Gamma \vdash_m e : t$.

$\underline{e = x}$**:** similar to the case of $e = c$.

$\underline{e = (e_1, e_2)}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*pair*) or (*subsum*). Assume that the typing derivation ends with (*pair*):

$$\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : t_1} \quad \dfrac{\dots}{\Delta \wr \Gamma \vdash e_2 : t_2}}{\Delta \wr \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

By **E**-induction, we have $\Delta \wr \Gamma \vdash_m e_i : t_i$. Then the rule (*pair*) gives us $\Delta \wr \Gamma \vdash_m (e_1, e_2) : t_1 \times t_2$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \pi_i(e')}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*proj*) or (*subsum*). Assume that the typing derivation ends with (*proj*):

$$\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e' : (t_1 \times t_2)}}{\Delta \wr \Gamma \vdash \pi_i(e') : t_i}\ (proj)$$

By **E**-induction, we have $\Delta \wr \Gamma \vdash_m e' : (t_1 \times t_2)$. Then the rule (*proj*) gives us $\Delta \wr \Gamma \vdash_m \pi_i(e') : t_i$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e_1 e_2}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*appl*) or (*subsum*). Assume that the typing derivation ends with (*appl*):

$$\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : t_1 \to t_2} \quad \dfrac{\dots}{\Delta \wr \Gamma \vdash e_2 : t_1}}{\Delta \wr \Gamma \vdash e_1 e_2 : t_2}\ (appl)$$

By **E**-induction, we have $\Delta \wr \Gamma \vdash_m e_1 : t_1 \to t_2$ and $\Delta \wr \Gamma \vdash_m e_2 : t_1$. Then the rule (*appl*) gives us $\Delta \wr \Gamma \vdash_m e_1 e_2 : t_2$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e'}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*abstr*) or (*subsum*). Assume that the typing derivation ends with (*abstr*):

$$\dfrac{\begin{array}{l}\forall i \in I, j \in J.\ \dfrac{\dots}{\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash e'@[\sigma_j] : s_i\sigma_j} \\ \Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j))\end{array}}{\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)}\ (abstr)$$

By **E**-induction, for all $i \in I$ and $j \in J$, we have $\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash_m e'@[\sigma_j] : s_i\sigma_j$. Then the rule (*abstr*) gives us $\Delta \wr \Gamma \vdash_m \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e' : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'\in t\,?\,e_1 : e_2}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*case*) or (*subsum*). Assume that the typing derivation ends with (*case*):

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_2 : s} \end{cases}}{\Delta \wr \Gamma \vdash (e'\in t\,?\,e_1 : e_2) : s}\ (case)$$

By **E**-induction, we have $\Delta \wr \Gamma \vdash_m e' : t'$ and $\Delta \wr \Gamma \vdash_m e_i : s$ (for i such that e_i has been effectively type-checked). Then the rule (*case*) gives us $\Delta \wr \Gamma_m \vdash (e'\in t\,?\,e_1 : e_2) : s$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma]}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*inst*) or (*subsum*). Assume that the typing derivation ends with (*inst*):

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : t} \quad \sigma \sharp \Delta}{\Delta \wr \Gamma \vdash e'[\sigma] : t\sigma}\ (inst)$$

By **E**-induction, we have $\Delta \wr \Gamma \vdash_m e' : t$. Since $\sigma \sharp \Delta$, applying (*instinter*) where $|J| = 1$, we get $\Delta \wr \Gamma \vdash_m e'[\sigma] : t\sigma$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

$\underline{e = e'[\sigma_j]_{j\in J}}$**:** the typing derivation $\Delta \wr \Gamma \vdash e : t$ should end with either (*inter*) or (*subsum*). Assume that the typing derivation ends with (*inter*):

$$\dfrac{\forall j \in J.\ \dfrac{\cdots}{\Delta \wr \Gamma \vdash e'[\sigma_j] : t_j} \quad |J| > 1}{\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (inter)$$

As an intermediary result, we first prove that the derivation can be rewritten as

$$\dfrac{\dfrac{\forall j \in J.\ \dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : s} \quad \sigma_j \sharp \Delta}{\Delta \wr \Gamma \vdash e'[\sigma_j] : s\sigma_j}\ (inst)}{\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j}\ (inter) \qquad \bigwedge_{j\in J} s\sigma_j \leq \bigwedge_{j\in J} t_j}{\Delta \wr \Gamma \vdash e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t_j}\ (subsum)$$

We proceed by induction on the original derivation. It is clear that each sub-derivation $\Delta \wr \Gamma \vdash e'[\sigma_j] : t_j$ ends with either (*inst*) or (*subsum*). If all the sub-derivations end with an instance of (*inst*), then for all $j \in J$, we have

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e' : s_j} \quad \sigma_j \sharp \Delta}{\Delta \wr \Gamma \vdash e'[\sigma_j] : s_j\sigma_j}\ (inst)$$

By Lemma B.2, we have $\Delta \wr \Gamma \vdash e' : \bigwedge_{j\in J} s_j$. Let $s = \bigwedge_{j\in J} s_j$. Then by (*inst*), we get $\Delta \wr \Gamma \vdash e'[\sigma_j] : s\sigma_j$. Finally, by (*inter*) and (*subsum*), the intermediary result holds. Otherwise, there is at least one of the sub-derivations ends with an instance of (*subsum*), the intermediary result also hold by induction.
Now that the intermediary result is proved, we go back to the proof of the lemma. By **E**-induction on e' (*i.e.*, $\Delta \wr \Gamma \vdash e' : s$), we have $\Delta \wr \Gamma \vdash_m e' : s$. Since $\sigma_j \sharp \Delta$, applying (*instinter*), we get $\Delta \wr \Gamma \vdash_m e'[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Finally, by subsumption, we get $\Delta \wr \Gamma \vdash_m e'[\sigma]_{j\in J} : \bigwedge_{j\in J} t_j$.
Otherwise, the typing derivation ends with an instance of (*subsum*), similar to the case of $e = c$, the result follows by **I**-induction.

□

From now on we will use $\vdash$ to denote $\vdash_m$, that is the system with the merged rule.

C.2 Algorithmic Typing Rules

In this section, we analyze the typing derivations produced by the rules of Section A.3 to see where subsumption is needed and where it can be pushed down the derivation tree. We need first some preliminary definitions and decomposition results about pair and function types to deal with the projection and application rules.

C.2.1 Pair types

A type s is a pair type if $s \leq \mathbb{1} \times \mathbb{1}$. If an expression e is typeable with a pair type s, we want to compute from s a valid type for $\pi_i(e)$. In $\mathbb{C}$Duce, a pair type s is a finite union of product types, which can be decomposed into a finite set of pairs of types, denoted as $\boldsymbol{\pi}(s)$. For example, we decompose $s = (t_1 \times t_2) \vee (s_1 \times s_2)$ as $\boldsymbol{\pi}(s) = \{(t_1, t_2), (s_1, s_2)\}$. We can then compute easily a type $\boldsymbol{\pi}_i(s)$ for $\pi_i(e)$ as $\boldsymbol{\pi}_i(s) = t_i \vee s_i$ (we used boldface symbols to distinguish these type operators from the projections used in expressions). In the calculus considered here, the situation becomes more complex because of type variables, especially top level ones. Let s be a pair type that contains a top-level variable α. Since $\alpha \not\leq \mathbb{1} \times \mathbb{1}$ and $s \leq \mathbb{1} \times \mathbb{1}$, then it is not possible that $s \simeq s' \vee \alpha$. In other terms the top-level variable cannot appear alone in a union: it must occur intersected with some product type so that it does not "overtake" the $\mathbb{1} \times \mathbb{1}$ bound. Consequently, we have $s \simeq s' \wedge \alpha$ for some $s' \leq \mathbb{1} \times \mathbb{1}$. However, in a typing derivation starting from $\Delta \fatsemi \Gamma \vdash e : s$ and ending with $\Delta \fatsemi \Gamma \vdash \pi_i(e) : t$, there exists an intermediary step where e is assigned a type of the form $(t_1 \times t_2)$ (and that verifies $s \leq (t_1 \times t_2)$) before applying the projection rule. So it is necessary to get rid of the top-level variables of s (using subsumption) before computing the projection. The example above shows that α does not play any role since it is the s' component that will be used to subsume s to a product type. To say it otherwise, since e has type s for all possible assignment of α, then the typing derivation must hold also for $\alpha = \mathbb{1}$. In whatever way we look at it, the top-level type variables are useless and can be safely discarded when decomposing s.

Given a type t, we write $\mathsf{dnf}(t)$ for a disjunctive normal form of t, which is defined in [6]. Formally, we define the decomposition of a pair type as follows:

Definition C.2. *Let τ be a disjunctive normal form such that $\tau \leq \mathbb{1} \times \mathbb{1}$. We define the* decomposition *of τ as follows:*

$$\begin{aligned}
&\boldsymbol{\pi}(\textstyle\bigvee_{i\in I} \tau_i) = \bigcup_{i\in I} \boldsymbol{\pi}(\tau_i)\\
&\boldsymbol{\pi}(\textstyle\bigwedge_{j\in P}(t_1^j \times t_2^j) \wedge \bigwedge_{k\in N} \neg(t_1^k \times t_2^k) \wedge \bigwedge_{\alpha\in P_\mathscr{V}} \alpha \wedge \bigwedge_{\alpha'\in N_\mathscr{V}} \neg\alpha') \quad (|P| > 0)\\
&\qquad = \boldsymbol{\pi}(\textstyle\bigvee_{N'\subseteq N}((\bigwedge_{j\in P} t_1^j \wedge \bigwedge_{k\in N'} \neg t_1^k) \times (\bigwedge_{j\in P} t_2^j \wedge \bigwedge_{k\in N\setminus N'} \neg t_2^k)))\\
&\boldsymbol{\pi}((t_1 \times t_2)) = \begin{cases} \{(t_1, t_2)\} & t_1 \not\simeq \mathbb{0} \text{ and } t_2 \not\simeq \mathbb{0} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

and the i-th projection as $\boldsymbol{\pi}_i(\tau) = \bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(\tau)} s_i$.
For all type t such that $t \leq \mathbb{1} \times \mathbb{1}$, the decomposition of t is defined as

$$\boldsymbol{\pi}(t) = \boldsymbol{\pi}(\mathsf{dnf}((\mathbb{1} \times \mathbb{1}) \wedge t))$$

and the i-th projection as $\boldsymbol{\pi}_i(t) = \bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(\mathsf{dnf}((\mathbb{1}\times\mathbb{1})\wedge t))} s_i$

The decomposition of a union of pair types is the union of each decomposition. When computing the decomposition of an intersection of product types and top-level type variables, we compute all the possible distributions of the intersections over the products, and we discard the top-level variables, as discussed above. Finally, the decomposition of a product is the pair of two components, provided that both components are not empty.

We now prove that the top-level type variables can be safely eliminated in a well-founded (convex) model with infinite support (see [6] for the definitions of model, convexity and infinite support).

Lemma C.3. *Let $\leq$ be the subtyping relation induced by a well-founded (convex) model with infinite support. Then*

$$\bigwedge_{p\in P} (t_p \times s_p) \wedge \alpha \leq \bigvee_{n\in N} (t_n \times s_n) \iff \bigwedge_{p\in P} (t_p \times s_p) \leq \bigvee_{n\in N} (t_n \times s_n)$$

Proof. The result trivially holds if $\bigwedge_{p\in P} (t_p \times s_p) = \mathbb{0}$ or $|P| = 0$ (*ie*, $\bigwedge_{p\in P} (t_p \times s_p) = \mathbb{1}$). Let us examine the remaining cases:

$\Leftarrow$: straightforward.

$\Rightarrow$: Assume that $\bigwedge_{p\in P} (t_p \times s_p) \not\leq \bigvee_{n\in N} (t_n \times s_n)$. Let τ be the type $\bigwedge_{p\in P} (t_p \times s_p) \wedge \bigwedge_{n\in N} \neg(t_n \times s_n)$. Then there exists an assignment η such that $[\![\tau]\!]\eta \neq \emptyset$ (see the subtyping relation defined in [6]). Using the procedure explore_pos defined in the proof of Lemma 3.23 in [6], we can generate an element d belonging to $[\![\tau]\!]\eta$.[8] The procedure explore_pos also generates an assignment η_0 for the type variables in $\mathsf{var}(\tau)$. We define η' such that $\eta'(\alpha) = \eta_0(\alpha) \cup \{d\}$, $\eta'(\neg\alpha) = \eta_0(\neg\alpha) \setminus \{d\}$, and $\eta' = \eta_0$ otherwise. Then we have $[\![\tau \wedge \alpha]\!]\eta' \neq \emptyset$, which implies $\bigwedge_{p\in P} (t_p \times s_p) \wedge \alpha \not\leq \bigvee_{n\in N} (t_n \times s_n)$. The result follows by the contrapositive.

□

The decomposition of pair types defined above has the following properties:

[8] Strictly speaking, the procedure explore_pos of Lemma 3.23 in [6] supposes τ contains only finite product types, but it can be extended to infinite product types by Lemma 3.24 in [6]

Lemma C.4. *Let $\leq$ be the subtyping relation induced by a well-founded (convex) model with infinite support and t a type such that $t \leq \mathbb{1} \times \mathbb{1}$. Then*

(1) *For all $(t_1, t_2) \in \boldsymbol{\pi}(t)$, we have $t_1 \not\simeq \mathbb{0}$ and $t_2 \not\simeq \mathbb{0}$*
(2) *For all s_1, s_2, we have $t \leq (s_1 \times s_2) \iff \bigvee_{(t_1,t_2)\in\boldsymbol{\pi}(t)}(t_1 \times t_2) \leq (s_1 \times s_2)$*

Proof.

(1)**:** straightforward.
(2)**:** Since $t \leq \mathbb{1} \times \mathbb{1}$, we have

$$t \simeq \bigvee_{(P,N)\in\mathsf{dnf}(t)} ((\mathbb{1} \times \mathbb{1}) \wedge \bigwedge_{j\in P\setminus\mathscr{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathscr{V}} \neg(t_1^k \times t_2^k) \wedge \bigwedge_{\alpha\in P\cap\mathscr{V}} \alpha \wedge \bigwedge_{\alpha'\in N\cap\mathscr{V}} \neg\alpha')$$

If $t \simeq \mathbb{0}$, then $\boldsymbol{\pi}(t) = \emptyset$, and the result holds. Assume that $t \not\simeq \mathbb{0}$, $|P| > 0$ and each summand of $\mathsf{dnf}(t)$ is not equivalent to $\mathbb{0}$ as well. Let $\ulcorner t \urcorner$ denote the type $\bigvee_{(P,N)\in\mathsf{dnf}(t)}(\bigwedge_{j\in P\setminus\mathscr{V}}(t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathscr{V}} \neg(t_1^k \times t_2^k))$. Using the set-theoretic interpretation of types we have that $\ulcorner t \urcorner$ is equivalent to

$$\bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigvee_{N'\subseteq N\setminus\mathscr{V}} \Big(\Big(\bigwedge_{j\in P\setminus\mathscr{V}} t_1^j \wedge \bigwedge_{k\in N'} \neg t_1^k \Big) \times \Big(\bigwedge_{j\in P\setminus\mathscr{V}} t_2^j \wedge \bigwedge_{k\in (N\setminus\mathscr{V})\setminus N'} \neg t_2^k \Big) \Big) \Big)$$

This means that, $\ulcorner t \urcorner$ is a equivalent to a union of product types. Let us rewrite this union more explicitly, that is, $\ulcorner t \urcorner \simeq \bigvee_{i\in I}(t_1^i \times t_2^i)$ obtained as follows

$$\bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigvee_{N'\subseteq N\setminus\mathscr{V}} \Big(\overbrace{\Big(\bigwedge_{j\in P\setminus\mathscr{V}} t_1^j \wedge \bigwedge_{k\in N'} \neg t_1^k \Big)}^{t_1^i} \times \overbrace{\Big(\bigwedge_{j\in P\setminus\mathscr{V}} t_2^j \wedge \bigwedge_{k\in (N\setminus\mathscr{V})\setminus N'} \neg t_2^k \Big)}^{t_2^i} \Big) \Big)$$

We have

$$\boldsymbol{\pi}(t) = \{(t_1^i, t_2^i) \mid i \in I \text{ and } t_1^i \not\simeq \mathbb{0} \text{ and } t_2^i \not\simeq \mathbb{0}\}$$

Finally, for all pair of types s_1 and s_2, we have

$$\begin{aligned}
& t \leq (s_1 \times s_2) \\
\iff & \bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigwedge_{j\in P\setminus\mathscr{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathscr{V}} \neg(t_1^k \times t_2^k) \wedge \bigwedge_{\alpha\in P\cap\mathscr{V}} \alpha \wedge \bigwedge_{\alpha'\in N\cap\mathscr{V}} \neg\alpha' \Big) \leq (s_1 \times s_2) \\
\iff & \bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigwedge_{j\in P\setminus\mathscr{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathscr{V}} \neg(t_1^k \times t_2^k) \wedge \bigwedge_{\alpha\in P\cap\mathscr{V}} \alpha \wedge \bigwedge_{\alpha'\in N\cap\mathscr{V}} \neg\alpha' \wedge \neg(s_1 \times s_2) \Big) \leq \mathbb{0} \\
\iff & \bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigwedge_{j\in P\setminus\mathscr{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathscr{V}} \neg(t_1^k \times t_2^k) \wedge \neg(s_1 \times s_2) \Big) \leq \mathbb{0} \quad \text{(Lemma C.3)} \\
\iff & \bigvee_{(P,N)\in\mathsf{dnf}(t)} \Big(\bigwedge_{j\in P\setminus\mathscr{V}} (t_1^j \times t_2^j) \wedge \bigwedge_{k\in N\setminus\mathscr{V}} \neg(t_1^k \times t_2^k) \Big) \leq (s_1 \times s_2) \\
\iff & \ulcorner t \urcorner \leq (s_1 \times s_2) \\
\iff & \bigvee_{(t_1,t_2)\in\boldsymbol{\pi}(t)} (t_1 \times t_2) \leq (s_1 \times s_2)
\end{aligned}$$

□

Lemma C.5. *Let s be a type such that $s \leq (t_1 \times t_2)$. Then*

(1) $s \leq (\boldsymbol{\pi}_1(s) \times \boldsymbol{\pi}_2(s))$
(2) $\boldsymbol{\pi}_i(s) \leq t_i$

Proof. (1)**:** according to the proof of Lemma C.4, $\bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(s)}(s_1 \times s_2)$ is equivalent to the type obtained from s by ignoring all the top-level type variables. Then it is trivial that $s \leq \bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(s)}(s_1 \times s_2)$ and then $s \leq (\boldsymbol{\pi}_1(s) \times \boldsymbol{\pi}_2(s))$.
(2)**:** since $s \leq (t_1 \times t_2)$, according to Lemma C.4, we have $\bigvee_{(s_1,s_2)\in\boldsymbol{\pi}(s)}(s_1 \times s_2) \leq (t_1 \times t_2)$. So for all $(s_1, s_2) \in \boldsymbol{\pi}(s)$, we have $(s_1 \times s_2) \leq (t_1 \times t_2)$. Moreover, as s_i is not empty, we have $s_i \leq t_i$. Therefore, $\boldsymbol{\pi}_i(s) \leq t_i$.

□

Lemma C.6. *Let t and s be two types such that $t \leq \mathbb{1} \times \mathbb{1}$ and $s \leq \mathbb{1} \times \mathbb{1}$. Then $\boldsymbol{\pi}_i(t \wedge s) \leq \boldsymbol{\pi}_i(t) \wedge \boldsymbol{\pi}_i(s)$.*

Proof. Let $t = \bigvee_{j_1\in J_1} \tau_{j_1}$ and $s = \bigvee_{j_2\in J_2} \tau_{j_2}$ such that

$$\tau_j = (t_j^1 \times t_j^2) \wedge \bigwedge_{\alpha\in P_j} \alpha \wedge \bigwedge_{\alpha'\in N_j} \neg\alpha'$$

and $\tau_j \not\simeq \mathbb{0}$ for all $j \in J_1 \cup J_2$. Then we have $t \wedge s = \bigvee_{j_1 \in J_1, j_2 \in J_2} \tau_{j_1} \wedge \tau_{j_2}$. Let $j_1 \in J_1$ and $j_2 \in J_2$. If $\tau_{j_1} \wedge \tau_{j_2} \simeq \mathbb{0}$, we have $\boldsymbol{\pi}_i(\tau_{j_1} \wedge \tau_{j_2}) = \mathbb{0}$. Otherwise, $\boldsymbol{\pi}_i(\tau_{j_1} \wedge \tau_{j_2}) = t^i_{j_1} \wedge t^i_{j_2} = \boldsymbol{\pi}_i(\tau_{j_1}) \wedge \boldsymbol{\pi}_i(\tau_{j_2})$. For both cases, we have $\boldsymbol{\pi}_i(\tau_{j_1} \wedge \tau_{j_2}) \leq \boldsymbol{\pi}_i(\tau_{j_1}) \wedge \boldsymbol{\pi}_i(\tau_{j_2})$. Therefore

$$\begin{array}{rcl} \boldsymbol{\pi}_i(t \wedge s) & \simeq & \bigvee_{j_1 \in J_1, j_2 \in J_2} \boldsymbol{\pi}_i(\tau_{j_1} \wedge \tau_{j_2}) \\ & \leq & \bigvee_{j_1 \in J_1, j_2 \in J_2} (\boldsymbol{\pi}_i(\tau_{j_1}) \wedge \boldsymbol{\pi}_i(\tau_{j_2})) \\ & \simeq & (\bigvee_{j_1 \in J_1} \boldsymbol{\pi}_i(\tau_{j_1})) \wedge (\bigvee_{j_2 \in J_2} \boldsymbol{\pi}_i(\tau_{j_2})) \\ & \simeq & \boldsymbol{\pi}_i(t) \wedge \boldsymbol{\pi}_i(s) \end{array}$$

□

For example, $\boldsymbol{\pi}_1((\texttt{Int} \times \texttt{Int}) \wedge (\texttt{Int} \times \texttt{Bool})) = \boldsymbol{\pi}_1(\mathbb{0}) = \mathbb{0}$, while $\boldsymbol{\pi}_1((\texttt{Int} \times \texttt{Int})) \wedge \boldsymbol{\pi}_1((\texttt{Int} \times \texttt{Bool})) = \texttt{Int} \wedge \texttt{Int} = \texttt{Int}$.

Lemma C.7. *Let t be a type and σ be a type substitution such that $t \leq \mathbb{1} \times \mathbb{1}$. Then $\boldsymbol{\pi}_i(t\sigma) \leq \boldsymbol{\pi}_i(t)\sigma$*

Proof. We put t into its disjunctive normal form $\bigvee_{j \in J} \tau_j$ such that

$$\tau_j = (t^1_j \times t^2_j) \wedge \bigwedge_{\alpha \in P_j} \alpha \wedge \bigwedge_{\alpha' \in N_j} \neg\alpha'$$

and $\tau_j \not\simeq \mathbb{0}$ for all $j \in J$. Then we have $t\sigma = \bigvee_{j \in J} \tau_j\sigma$. So $\boldsymbol{\pi}_i(t\sigma) = \bigvee_{j \in J} \boldsymbol{\pi}_i(\tau_j\sigma)$. Let $j \in J$. If $\tau_j\sigma \simeq \mathbb{0}$, then $\boldsymbol{\pi}_i(\tau_j\sigma) = \mathbb{0}$ and trivially $\boldsymbol{\pi}_i(\tau_j\sigma) \leq \boldsymbol{\pi}_i(\tau_j)\sigma$. Otherwise, we have $\tau_j\sigma = (t^1_j\sigma \times t^2_j\sigma) \wedge (\bigwedge_{\alpha \in P_j} \alpha \wedge \bigwedge_{\alpha' \in N_j} \neg\alpha')\sigma$. By Lemma C.6, we get $\boldsymbol{\pi}_i(\tau_j\sigma) \leq t^i_j\sigma \wedge \boldsymbol{\pi}_i((\bigwedge_{\alpha \in P_j} \alpha \wedge \bigwedge_{\alpha' \in N_j} \neg\alpha')\sigma) \leq t^i_j\sigma \simeq \boldsymbol{\pi}_i(\tau_j)\sigma$. Therefore, $\bigvee_{j \in J} \boldsymbol{\pi}_i(\tau_j\sigma) \leq \bigvee_{j \in J} \boldsymbol{\pi}_i(\tau_j)\sigma$, that is, $\boldsymbol{\pi}_i(t\sigma) \leq \boldsymbol{\pi}_i(t)\sigma$. □

For example, $\boldsymbol{\pi}_1(((\texttt{Int} \times \texttt{Int}) \wedge \alpha)\{(\texttt{Int} \times \texttt{Bool})/\alpha\}) = \boldsymbol{\pi}_1((\texttt{Int} \times \texttt{Int}) \wedge (\texttt{Int} \times \texttt{Bool})) = \mathbb{0}$, while $(\boldsymbol{\pi}_1((\texttt{Int} \times \texttt{Int})))\{(\texttt{Int} \times \texttt{Bool})/\alpha\} = \texttt{Int}\{(\texttt{Int} \times \texttt{Bool})/\alpha\} = \texttt{Int}$.

Lemma C.8. *Let t be a type such that $t \leq \mathbb{1} \times \mathbb{1}$ and $[\sigma_k]_{k \in K}$ be a set of type substitutions. Then $\boldsymbol{\pi}_i(\bigwedge_{k \in K} t\sigma_k) \leq \bigwedge_{k \in K} \boldsymbol{\pi}_i(t)\sigma_k$*

Proof. Consequence of Lemmas C.6 and C.7. □

C.2.2 Function types

A type t is a function type if $t \leq \mathbb{0} \to \mathbb{1}$. In order to type the application of a function having a function type t, we need to determine the domain of t, that is, the set of values the function can be safely applied to. This problem has been solved for ground function types in [12]. Again, the problem becomes more complex if t contains top-level type variables. Another issue is to determine what is the result type of an application of a function type t to an argument of type s (where s belongs to the domain of t), knowing that both t and s may contain type variables.

Following the same reasoning as with pair types, if a function type t contains a top-level variable α, then $t \simeq t' \wedge \alpha$ for some function type t'. In a typing derivation for a judgment $\Delta \fatsemi \Gamma \vdash e_1\ e_2 : t$ which contains $\Delta \fatsemi \Gamma \vdash e_1 : t$, there exists an intermediary step where we assign a type $t_1 \to t_2$ to e_1 (with $t \leq t_1 \to t_2$) before using the application rule. It is therefore necessary to eliminate the top-level variables from the function type t before we can type an application. Once more, the top-level variables are useless when computing the domain of t and can be safely discarded.

Formally, we define the domain of a function type as follows:

Definition C.9 (Domain). *Let τ be a disjunctive normal form such that $\tau \leq \mathbb{0} \to \mathbb{1}$. We define* $\mathsf{dom}(\tau)$*, the* domain *of τ, as:*

$$\begin{array}{l} \mathsf{dom}(\bigvee_{i \in I} \tau_i) = \bigwedge_{i \in I} \mathsf{dom}(\tau_i) \\ \mathsf{dom}(\bigwedge_{j \in P}(t^j_1 \to t^j_2) \wedge \bigwedge_{k \in N} \neg(t^k_1 \to t^k_2) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha') \\ \qquad = \begin{cases} \mathbb{1} & \text{if } \bigwedge_{j \in P}(t^j_1 \to t^j_2) \wedge \bigwedge_{k \in N} \neg(t^k_1 \to t^k_2) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha' \simeq \mathbb{0} \\ \bigvee_{j \in P} t^j_1 & \text{otherwise} \end{cases} \end{array}$$

For any type t such that $t \leq \mathbb{0} \to \mathbb{1}$, the domain of t is defined as

$$\mathsf{dom}(t) = \mathsf{dom}(\mathsf{dnf}((\mathbb{0} \to \mathbb{1}) \wedge t))$$

We also define a decomposition operator $\boldsymbol{\phi}$ that —akin to the decomposition operator $\boldsymbol{\pi}$ for product types— decomposes a function type into a finite set of pairs:

Definition C.10. *Let τ be a disjunctive normal form such that $\tau \leq \mathbb{0} \to \mathbb{1}$. We define the* decomposition *of τ as:*

$$\begin{array}{l} \boldsymbol{\phi}(\bigvee_{i \in I} \tau_i) = \bigcup_{i \in I} \boldsymbol{\phi}(\tau_i) \\ \boldsymbol{\phi}(\bigwedge_{j \in P}(t^j_1 \to t^j_2) \wedge \bigwedge_{k \in N} \neg(t^k_1 \to t^k_2) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha') \\ \qquad = \begin{cases} \emptyset & \text{if } \bigwedge_{j \in P}(t^j_1 \to t^j_2) \wedge \bigwedge_{k \in N} \neg(t^k_1 \to t^k_2) \wedge \bigwedge_{\alpha \in P_\mathcal{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathcal{V}} \neg\alpha' \simeq \mathbb{0} \\ \{(\bigvee_{j \in P'} t^j_1, \bigwedge_{j \in P \setminus P'} t^j_2) \mid P' \subsetneq P\} & \text{otherwise} \end{cases} \end{array}$$

For any type t such that $t \leq \mathbb{0} \to \mathbb{1}$, the decomposition of $t$is defined as

$$\boldsymbol{\phi}(t) = \boldsymbol{\phi}(\mathsf{dnf}((\mathbb{0} \to \mathbb{1}) \wedge t)).$$

The set $\boldsymbol{\phi}(t)$ satisfies the following fundamental property: for every arrow type $s \to s'$, the constraint $t \leq s \to s'$ holds if and only if $s \leq \mathsf{dom}(t)$ holds and for all $(t_1, t_2) \in \boldsymbol{\phi}(t)$, either $s \leq t_1$ or $t_2 \leq s'$ hold (see Lemma C.12). As a result, the minimum type

$$t \cdot s = \min\{s' \mid t \leq s \to s'\}$$

exists, and it is defined as the union of all t_2 such that $s \not\leq t_1$ and $(t_1, t_2) \in \boldsymbol{\phi}(t)$ (see Lemma C.13). The type $t \cdot s$ is used to type the application of an expression of type t to an expression of type s.

As with pair types, in a well-founded (convex) model with infinite support, we can safely eliminate the top-level type variables.

Lemma C.11. *Let $\leq$ be the subtyping relation induced by a well-founded (convex) model with infinite support. Then*

$$\bigwedge_{p \in P} (t_p \to s_p) \wedge \alpha \leq \bigvee_{n \in N} (t_n \to s_n) \iff \bigwedge_{p \in P} (t_p \to s_p) \leq \bigvee_{n \in N} (t_n \to s_n)$$

Proof. Similar to the proof of Lemma C.3. □

Lemma C.12. *Let $\leq$ be the subtyping relation induced by a well-founded (convex) model with infinite support and t a type such that $t \leq \mathbb{0} \to \mathbb{1}$. Then*

$$\forall s_1, s_2 \,.\, (t \leq s_1 \to s_2) \iff \begin{cases} s_1 \leq \mathsf{dom}(t) \\ \forall (t_1, t_2) \in \boldsymbol{\phi}(t) \,.\, (s_1 \leq t_1) \text{ or } (t_2 \leq s_2) \end{cases}$$

Proof. Since $t \leq \mathbb{0} \to \mathbb{1}$, we have

$$t \simeq \bigvee_{(P,N) \in \mathsf{dnf}(t)} ((\mathbb{0} \to \mathbb{1}) \wedge \bigwedge_{j \in P \setminus \mathscr{V}} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N \setminus \mathscr{V}} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P \cap \mathscr{V}} \alpha \wedge \bigwedge_{\alpha' \in N \cap \mathscr{V}} \neg\alpha')$$

If $t \simeq \mathbb{0}$, then $\mathsf{dom}(t) = \mathbb{1}$, $\boldsymbol{\phi}(t) = \emptyset$, and the result holds. If $t \simeq t_1 \vee t_2$, then $t_1 \leq \mathbb{0} \to \mathbb{1}$, $t_2 \leq \mathbb{0} \to \mathbb{1}$, $\mathsf{dom}(t) = \mathsf{dom}(t_1) \wedge \mathsf{dom}(t_2)$ and $\boldsymbol{\phi}(t) = \boldsymbol{\phi}(t_1) \cup \boldsymbol{\phi}(t_2)$. So the result follows if it also holds for t_1 and t_2. Thus, without loss of generality, we can assume that t has the following form:

$$t \simeq \bigwedge_{j \in P} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P_{\mathscr{V}}} \alpha \wedge \bigwedge_{\alpha' \in N_{\mathscr{V}}} \neg\alpha'$$

where $P \neq \emptyset$ and $t \not\simeq \mathbb{0}$. Then $\mathsf{dom}(t) = \bigvee_{j \in P} t_1^j$ and $\boldsymbol{\phi}(t) = \{(\bigvee_{j \in P'} t_1^j, \bigwedge_{j \in P \setminus P'} t_2^j) \mid P' \subsetneq P\}$. For every pair of types s_1 and s_2, we have

$$\begin{aligned}
& t \leq (s_1 \to s_2) \\
\iff \quad & \bigwedge_{j \in P} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P_{\mathscr{V}}} \alpha \wedge \bigwedge_{\alpha' \in N_{\mathscr{V}}} \neg\alpha' \leq (s_1 \to s_2) \\
\iff \quad & \bigwedge_{j \in P} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \leq (s_1 \to s_2) \quad \text{(Lemma C.11)} \\
\iff \quad & \bigwedge_{j \in P} (t_1^j \to t_2^j) \leq s_1 \to s_2 \quad (\ulcorner t \urcorner \not\simeq \mathbb{0} \text{ and Lemma 3.12 in [6]}) \\
\iff \quad & \forall P' \subseteq P. \left(s_1 \leq \bigvee_{j \in P'} t_1^j \right) \vee \left(P \neq P' \wedge \bigwedge_{j \in P \setminus P'} t_2^j \leq s_2 \right)
\end{aligned}$$

where $\ulcorner t \urcorner = \bigwedge_{j \in P} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k)$.

□

Lemma C.13. *Let t and s be two types. If $t \leq s \to \mathbb{1}$, then $t \leq s \to s'$ has a smallest solution s', which is denoted as $t \cdot s$.*

Proof. Since $t \leq s \to \mathbb{1}$, by Lemma C.12, we have $s \leq \mathsf{dom}(t)$. Then the assertion $t \leq s \to s'$ is equivalent to:

$$\forall (t_1, t_2) \in \boldsymbol{\phi}(t). \, (s \leq t_1) \mathsf{or} (t_2 \leq s')$$

that is:

$$\left(\bigvee_{(t_1,t_2) \in \boldsymbol{\phi}(t) \text{ s.t. } (s \not\leq t_1)} t_2 \right) \leq s'$$

Thus the type $\bigvee_{(t_1,t_2) \in \boldsymbol{\phi}(t) \text{ s.t. } (s \not\leq t_1)} t_2$ is a lower bound for all the solutions.

By the subtyping relation on arrows it is also a solution, so it is the smallest one. To conclude, it suffices to take it as the definition for $t \cdot s$. □

We now prove some properties of the operators $\mathsf{dom}(\_)$ and "$\_ \cdot \_$".

Lemma C.14. *Let t be a type such that $t \leq \mathbb{0} \to \mathbb{1}$ and t', s, s' be types. Then*

(1) *if $s' \leq s \leq \mathsf{dom}(t)$, then $t \cdot s' \leq t \cdot s$.*
(2) *if $t' \leq t$, $s \leq \mathsf{dom}(t')$ and $s \leq \mathsf{dom}(t)$, then $t' \cdot s \leq t \cdot s$.*

Proof. (1) Since $s' \leq s$, we have $s \to t \cdot s \leq s' \to t \cdot s$. By definition of $t \cdot s$, we have $t \leq s \to t \cdot s$, therefore $t \leq s' \to t \cdot s$ holds. Consequently, we have $t \cdot s' \leq t \cdot s$ by definition of $t \cdot s'$.

(2) By definition, we have $t \leq s \to t \cdot s$, which implies $t' \leq s \to t \cdot s$. Therefore, $t \cdot s$ is a solution to $t' \leq s \to s'$, hence we have $t' \cdot s \leq t \cdot s$.

□

Lemma C.15. *Let t and s be two types such that $t \leq \mathbb{0} \to \mathbb{1}$ and $s \leq \mathbb{0} \to \mathbb{1}$. Then $\mathsf{dom}(t) \vee \mathsf{dom}(s) \leq \mathsf{dom}(t \wedge s)$.*

Proof. Let $t = \bigvee_{i_1 \in I_1} \tau_{i_1}$ and $s = \bigvee_{i_2 \in I_2} \tau_{i_2}$ such that $\tau_i \not\simeq \mathbb{0}$ for all $i \in I_1 \cup I_2$. Then we have $t \wedge s = \bigvee_{i_1 \in I_1, i_2 \in I_2} \tau_{i_1} \wedge \tau_{i_2}$. Let $i_1 \in I_1$ and $i_2 \in I_2$. If $\tau_{i_1} \wedge \tau_{i_2} \simeq \mathbb{0}$, then $\mathsf{dom}(\tau_{i_1} \wedge \tau_{i_2}) = \mathbb{1}$. Otherwise, $\mathsf{dom}(\tau_{i_1} \wedge \tau_{i_2}) = \mathsf{dom}(\tau_{i_1}) \vee \mathsf{dom}(\tau_{i_2})$. In both cases, we have $\mathsf{dom}(\tau_{i_1} \wedge \tau_{i_2}) \geq \mathsf{dom}(\tau_{i_1}) \vee \mathsf{dom}(\tau_{i_2})$. Therefore

$$\begin{array}{rl} \mathsf{dom}(t \wedge s) \simeq & \bigwedge_{i_1 \in I_1, i_2 \in I_2} \mathsf{dom}(\tau_{i_1} \wedge \tau_{i_2}) \\ \geq & \bigwedge_{i_1 \in I_1, i_2 \in I_2} (\mathsf{dom}(\tau_{i_1}) \vee \mathsf{dom}(\tau_{i_2})) \\ \simeq & \bigwedge_{i_1 \in I_1} (\bigwedge_{i_2 \in I_2} (\mathsf{dom}(\tau_{i_1}) \vee \mathsf{dom}(\tau_{i_2}))) \\ \simeq & \bigwedge_{i_1 \in I_1} (\mathsf{dom}(\tau_{i_1}) \vee (\bigwedge_{i_2 \in I_2} \mathsf{dom}(\tau_{i_2}))) \\ \geq & \bigwedge_{i_1 \in I_1} (\mathsf{dom}(\tau_{i_1})) \\ \simeq & \mathsf{dom}(t) \end{array}$$

Similarly, $\mathsf{dom}(t \wedge s) \geq \mathsf{dom}(s)$. Therefore $\mathsf{dom}(t) \vee \mathsf{dom}(s) \leq \mathsf{dom}(t \wedge s)$. □

For example, $\mathsf{dom}((\mathtt{Int} \to \mathtt{Int}) \wedge \neg(\mathtt{Bool} \to \mathtt{Bool})) \vee \mathsf{dom}(\mathtt{Bool} \to \mathtt{Bool}) = \mathtt{Int} \vee \mathtt{Bool}$, while $\mathsf{dom}(((\mathtt{Int} \to \mathtt{Int}) \wedge \neg(\mathtt{Bool} \to \mathtt{Bool})) \wedge (\mathtt{Bool} \to \mathtt{Bool})) = \mathsf{dom}(\mathbb{0}) = \mathbb{1}$.

Lemma C.16. *Let t be a type and σ be a type substitution such that $t \leq \mathbb{0} \to \mathbb{1}$. Then $\mathsf{dom}(t)\sigma \leq \mathsf{dom}(t\sigma)$*

Proof. We put t into its disjunctive normal form $\bigvee_{i \in I} \tau_i$ such that $\tau_i \not\simeq \mathbb{0}$ for all $i \in I$. Then we have $t\sigma = \bigvee_{i \in I} \tau_i\sigma$. So $\mathsf{dom}(t\sigma) = \bigwedge_{i \in I} \mathsf{dom}(\tau_i\sigma)$. Let $i \in I$. If $\tau_i\sigma \simeq \mathbb{0}$, then $\mathsf{dom}(\tau_i\sigma) = \mathbb{1}$. Otherwise, let $\tau_i = \bigwedge_{j \in P} (t_1^j \to t_2^j) \wedge \bigwedge_{k \in N} \neg(t_1^k \to t_2^k) \wedge \bigwedge_{\alpha \in P_\mathscr{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathscr{V}} \neg\alpha'$. Then $\mathsf{dom}(\tau_i) = \bigvee_{j \in P} t_1^j$ and $\mathsf{dom}(\tau_i\sigma) = \bigvee_{j \in P} t_1^j\sigma \vee \mathsf{dom}((\bigwedge_{\alpha \in P_\mathscr{V}} \alpha \wedge \bigwedge_{\alpha' \in N_\mathscr{V}} \neg\alpha')\sigma \wedge \mathbb{0} \to \mathbb{1})$. In both cases, we have $\mathsf{dom}(\tau_i)\sigma \leq \mathsf{dom}(\tau_i\sigma)$. Therefore, $\bigwedge_{i \in I} \mathsf{dom}(\tau_i)\sigma \leq \bigwedge_{i \in I} \mathsf{dom}(\tau_i\sigma)$, that is, $\mathsf{dom}(t)\sigma \leq \mathsf{dom}(t\sigma)$. □

For example, $\mathsf{dom}((\mathtt{Int} \to \mathtt{Int}) \wedge \neg\alpha)\{(\mathtt{Int} \to \mathtt{Int})/\alpha\} = \mathtt{Int}\{(\mathtt{Int} \to \mathtt{Int})/\alpha\} = \mathtt{Int}$, while $\mathsf{dom}(((\mathtt{Int} \to \mathtt{Int}) \wedge \neg\alpha)\{(\mathtt{Int} \to \mathtt{Int})/\alpha\}) = \mathsf{dom}((\mathtt{Int} \to \mathtt{Int}) \wedge \neg(\mathtt{Int} \to \mathtt{Int})) = \mathbb{1}$.

Lemma C.17. *Let t be a type such that $t \leq \mathbb{0} \to \mathbb{1}$ and $[\sigma_k]_{k \in K}$ be a set of type substitutions. Then $\bigwedge_{k \in K} \mathsf{dom}(t)\sigma_k \leq \mathsf{dom}(\bigwedge_{k \in K} t\sigma_k)$*

Proof.

$$\begin{array}{rll} \bigwedge_{k \in K} \mathsf{dom}(t)\sigma_k \leq & \bigwedge_{k \in K} \mathsf{dom}(t\sigma_k) & \text{(by Lemma C.16)} \\ \leq & \bigvee_{k \in K} \mathsf{dom}(t\sigma_k) & \\ \leq & \mathsf{dom}(\bigwedge_{k \in K} t\sigma_k) & \text{(by Lemma C.15)} \end{array}$$

□

Lemma C.18. *Let t_1, s_1, t_2 and s_2 be types such that $t_1 \cdot s_1$ and $t_2 \cdot s_2$ exists. Then $(t_1 \wedge t_2) \cdot (s_1 \wedge s_2)$ exists and $(t_1 \wedge t_2) \cdot (s_1 \wedge s_2) \leq (t_1 \cdot s_1) \wedge (t_2 \cdot s_2)$.*

Proof. According to Lemma C.13, we have $s_i \leq \mathsf{dom}(t_i)$ and $t_i \leq s_i \to (t_i \cdot s_i)$. Then by Lemma C.15, we get $s_1 \wedge s_2 \leq \mathsf{dom}(t_1) \wedge \mathsf{dom}(t_2) \leq \mathsf{dom}(t_1 \wedge t_2)$. Moreover, $t_1 \wedge t_2 \leq (s_1 \to (t_1 \cdot s_1)) \wedge (s_2 \to (t_2 \cdot s_2)) \leq (s_1 \wedge s_2) \to ((t_1 \cdot s_1) \wedge (t_2 \cdot s_2))$. Therefore, $(t_1 \wedge t_2) \cdot (s_1 \wedge s_2)$ exists and $(t_1 \wedge t_2) \cdot (s_1 \wedge s_2) \leq (t_1 \cdot s_1) \wedge (t_2 \cdot s_2)$. □

For example, $((\mathtt{Int} \to \mathtt{Bool}) \wedge (\mathtt{Bool} \to \mathtt{Bool})) \cdot (\mathtt{Int} \wedge \mathtt{Bool}) = \mathbb{0}$, while $((\mathtt{Int} \to \mathtt{Bool}) \cdot \mathtt{Int}) \wedge ((\mathtt{Bool} \to \mathtt{Bool}) \cdot \mathtt{Bool}) = \mathtt{Bool} \wedge \mathtt{Bool} = \mathtt{Bool}$.

Lemma C.19. *Let t and s be two types such that $t \cdot s$ exists and σ be a type substitution. Then $(t\sigma) \cdot (s\sigma)$ exists and $(t\sigma) \cdot (s\sigma) \leq (t \cdot s)\sigma$.*

Proof. Because $t \cdot s$ exists, we have $s \leq \mathsf{dom}(t)$ and $t \leq s \to (t \cdot s)$. Then $s\sigma \leq \mathsf{dom}(t)\sigma$ and $t\sigma \leq s\sigma \to (t \cdot s)\sigma$. By Lemma C.16, we get $\mathsf{dom}(t)\sigma \leq \mathsf{dom}(t\sigma)$. So $s\sigma \leq \mathsf{dom}(t\sigma)$. Therefore, $(t\sigma) \cdot (s\sigma)$ exists. Moreover, since $(t \cdot s)\sigma$ is a solution to $t\sigma \leq s\sigma \to s'$, by definition, we have $(t\sigma) \cdot (s\sigma) \leq (t \cdot s)\sigma$. □

For example, $(((\texttt{Int} \to \texttt{Int}) \wedge \neg\alpha)\sigma) \cdot (\texttt{Int}\sigma) = \mathbb{0} \cdot \texttt{Int} = \mathbb{0}$, while $(((\texttt{Int} \to \texttt{Int}) \wedge \neg\alpha) \cdot \texttt{Int})\sigma = \texttt{Int}\sigma = \texttt{Int}$, where $\sigma = \{^{(\texttt{Int} \to \texttt{Int})}/_{\alpha}\}$.

Lemma C.20. *Let t and s be two types and $[\sigma_k]_{k\in K}$ be a set of type substitutions such that $t \cdot s$ exists. Then $(\bigwedge_{k\in K} t\sigma_k) \cdot (\bigwedge_{k\in K} s\sigma_k)$ exists and $(\bigwedge_{k\in K} t\sigma_k) \cdot (\bigwedge_{k\in K} s\sigma_k) \leq \bigwedge_{k\in K}(t \cdot s)\sigma_k$.*

Proof. According to Lemmas C.19 and C.18, $(\bigwedge_{k\in K} t\sigma_k) \cdot (\bigwedge_{k\in K} s\sigma_k)$ exists. Moreover,

$$\begin{array}{rll} \bigwedge_{k\in K}(t \cdot s)\sigma_k \geq & \bigwedge_{k\in K}(t\sigma_k \cdot s\sigma_k) & \text{(Lemma C.19)} \\ \geq & (\bigwedge_{k\in K} t\sigma_k) \cdot (\bigwedge_{k\in K} s\sigma_k) & \text{(Lemma C.18)} \end{array}$$

□

C.2.3 Syntax-Directed Rules

Because of subsumption, the typing rules provided in Section A.3 are not syntax-directed and so they do not yield a type-checking algorithm directly. In simply type λ-calculus, subsumption is used to bridge gaps between the types expected by functions and the actual types of their arguments in applications [19]. In our calculus, we identify four situations where the subsumption is needed, namely, the rules for projections, abstractions, applications, and type cases. To see why, we consider a typing derivation ending with each typing rule whose immediate sub-derivation ends with (*subsum*). For each case, we explain how the use of subsumption can be pushed through the typing rule under consideration, or how the rule should be modified to take subtyping into account.

First we consider the case where a typing derivation ends with (*subsum*) whose immediate sub-derivation also ends with (*subsum*). The two consecutive uses of (*subsum*) can be merged into one, because the subtyping relation is transitive.

Lemma C.21. *If $\Delta \wr \Gamma \vdash e : t$, then there exists a derivation for $\Delta \wr \Gamma \vdash e : t$ where there are no consecutive instances of (subsum).*

Proof. Assume that there exist two consecutive instances of (*subsum*) occurring in a derivation of $\Delta \wr \Gamma \vdash e : t$, that is,

$$\dfrac{\dots \quad \dfrac{\dfrac{\dots}{\Delta' \wr \Gamma' \vdash e' : s_2'} \quad s_2' \leq s_1'}{\Delta' \wr \Gamma' \vdash e' : s_1'}\,(subsum) \quad s_1' \leq t'}{\Delta' \wr \Gamma' \vdash e' : t'}\,(subsum) \quad \dots}{\Delta \wr \Gamma \vdash e : t}$$

Since $s_2' \leq s_1'$ and $s_1' \leq t'$, we have $s_2' \leq t'$. So we can rewrite this derivation as follows:

$$\dfrac{\dots \quad \dfrac{\dfrac{\dots}{\Delta' \wr \Gamma' \vdash e' : s_2'} \quad s_2' \leq t'}{\Delta' \wr \Gamma' \vdash e' : t'}\,(subsum) \quad \dots}{\Delta \wr \Gamma \vdash e : t}$$

Therefore, the result follows. □

Next, consider an instance of (*pair*) such that one of its sub-derivations ends with an instance of (*subsum*), for example, the left sub-derivation:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : s_1} \quad s_1 \leq t_1}{\Delta \wr \Gamma \vdash e_1 : t_1}\,(subsum) \quad \dfrac{\dots}{\Delta \wr \Gamma \vdash e_2 : t_2}}{\Delta \wr \Gamma \vdash (e_1, e_2) : (t_1 \times t_2)}\,(pair)$$

As $s_1 \leq t_1$, we have $s_1 \times t_2 \leq t_1 \times t_2$. Then we can move subsumption down through the rule (*pair*), giving the following derivation:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : s_1} \quad \dfrac{\dots}{\Delta \wr \Gamma \vdash e_2 : t_2}}{\Delta \wr \Gamma \vdash (e_1, e_2) : (s_1 \times t_2)}\,(pair) \quad s_1 \times t_2 \leq t_1 \times t_2}{\Delta \wr \Gamma \vdash (e_1, e_2) : (t_1 \times t_2)}\,(subsum)$$

The rule (*proj*) is a little trickier than (*pair*). Consider the following derivation:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e : s} \quad s \leq t_1 \times t_2}{\Delta \wr \Gamma \vdash e : (t_1 \times t_2)}\,(subsum)}{\Delta \wr \Gamma \vdash \pi_i(e) : t_i}\,(proj)$$

As $s \leq t_1 \times t_2$, s is a pair type. According to the decomposition of s and Lemma C.5, we can rewrite the previous derivation into the following one:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e : s} \quad s \leq \mathbb{1} \times \mathbb{1}}{\Delta \wr \Gamma \vdash \pi_i(e) : \boldsymbol{\pi}_i(s)} \quad \boldsymbol{\pi}_i(s) \leq t_i}{\Delta \wr \Gamma \vdash \pi_i(e) : t_i}$$

Note that the subtyping check $s \leq \mathbb{1} \times \mathbb{1}$ ensures that s is a pair type.

Next consider an instance of (*abstr*) (where $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j))$. All the sub-derivations may end with (*subsum*):

$$\dfrac{\forall i \in I, j \in J. \quad \dfrac{\dfrac{\dots}{\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s'_{ij}} \quad s'_{ij} \leq s_i\sigma_j}{\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j}}{\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)} \ (abstr)$$

Without subsumption, we would assign the type $\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s'_{ij})$ to the abstraction, while we want to assign the type $\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)$ to it because of the type annotations. Consequently, we have to keep the subtyping checks $s'_{ij} \leq s_i\sigma_j$ as side-conditions of an algorithmic typing rule for abstractions.

$$\dfrac{\forall i \in I, j \in J. \quad \Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s'_{ij} \qquad s'_{ij} \leq s_i\sigma_j}{\Delta \wr \Gamma \vdash \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)}$$

In (*appl*) case, suppose that both sub-derivations end with (*subsum*):

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : t} \quad t \leq t' \to s'}{\Delta \wr \Gamma \vdash e_1 : t' \to s'} \qquad \dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : s} \quad s \leq t'}{\Delta \wr \Gamma \vdash e_2 : t'}}{\Delta \wr \Gamma \vdash e_1\ e_2 : s'} \ (appl)$$

Since $s \leq t'$, then by the contravariance of arrow types we have $t' \to s' \leq s \to s'$. Hence, such a derivation can be rewritten as

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : t} \quad t \leq t' \to s'}{\Delta \wr \Gamma \vdash e_1 : t' \to s'} \qquad t' \to s' \leq s \to s'}{\Delta \wr \Gamma \vdash e_1 : s \to s'} \qquad \dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : s}}{\Delta \wr \Gamma \vdash e_1\ e_2 : s'} \ (appl)$$

Applying Lemma C.21, we can merge the two adjacent instances of (*subsum*) into one:

$$\dfrac{\dfrac{\dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : t} \quad t \leq s \to s'}{\Delta \wr \Gamma \vdash e_1 : s \to s'} \qquad \dfrac{\dots}{\Delta \wr \Gamma \vdash e_1 : s}}{\Delta \wr \Gamma \vdash e_1\ e_2 : s'} \ (appl)$$

A syntax-directed typing rule for applications can then be written as follows

$$\dfrac{\Delta \wr \Gamma \vdash e_1 : t \quad \Delta \wr \Gamma \vdash e_2 : s \quad t \leq s \to s'}{\Delta \wr \Gamma \vdash e_1 e_2 : s'}$$

where subsumption is used as a side condition to bridge the gap between the function type and the argument type.

This typing rule is not algorithmic yet, because the result type s' can be any type verifying the side condition. Using Lemma C.12, we can equivalently rewrite the side condition as $t \leq \mathbb{0} \to \mathbb{1}$ and $s \leq \mathsf{dom}(t)$ without involving the result type s'. The first condition ensures that t is a function type and the second one that the argument type s can be safely applied by t. Moreover, we assign the type $t \cdot s$ to the application, which is by definition the smallest possible type for it. We obtain then the following algorithmic typing rule.

$$\dfrac{\Delta \wr \Gamma \vdash e_1 : t \quad \Delta \wr \Gamma \vdash e_2 : s \quad t \leq \mathbb{0} \to \mathbb{1} \quad s \leq \mathsf{dom}(t)}{\Delta \wr \Gamma \vdash e_1 e_2 : t \cdot s}$$

Next, let us discuss the rule (*case*):

$$\dfrac{\Delta \wr \Gamma \vdash e : t' \qquad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \Delta \wr \Gamma \vdash e_1 : s \\ t' \not\leq t & \Rightarrow \quad \Delta \wr \Gamma \vdash e_2 : s \end{cases}}{\Delta \wr \Gamma \vdash (e \in t\, ?\, e_1 : e_2) : s} \ (case)$$

The rule covers four different situations, depending on which branches of the type-cases are checked: (i) no branch is type-checked, (ii) the first branch e_1 is type-checked, (iii) the second branch e_2 is type-checked, and (iv) both branches are type-checked. Each case produces a corresponding algorithmic rule.

In case (i), we have simultaneously $t' \leq t$ and $t' \leq \neg t$, which means that $t' = \mathbb{0}$. Consequently, e does not reduce to a value (otherwise, subject reduction would be violated), and neither does the whole type case.

Consequently, we can assign type $\mathbb{0}$ to the whole type.

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e : \mathbb{0}}}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : \mathbb{0}}$$

Suppose we are in case (ii) and the sub-derivation for the first branch e_1 ends with (*subsum*):

$$\dfrac{\Delta \wr \Gamma \vdash e : t' \quad t' \leq t \qquad \dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1 : s_1} \quad s_1 \leq s}{\Delta \wr \Gamma \vdash e_1 : s}}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s}\ (case)$$

Such a derivation can be rearranged as:

$$\dfrac{\dfrac{\Delta \wr \Gamma \vdash e : t' \quad t' \leq t \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1 : s_1}}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s_1} \quad s_1 \leq s}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s}$$

Moreover, (*subsum*) might also be used at the end of the sub-derivation for e:

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e : t''} \quad t'' \leq t'}{\Delta \wr \Gamma \vdash e : t'} \qquad t' \leq t \quad \Delta \wr \Gamma \vdash e_1 : s}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s}\ (case)$$

From $t'' \leq t'$ and $t' \leq t$, we deduce $t'' \leq t$ by transitivity. Therefore this use of subtyping can be merged with the subtyping check of the type case rule. We then obtain the following algorithmic rule.

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e : t''} \quad t'' \leq t \quad \Delta \wr \Gamma \vdash e_1 : s}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s}$$

We obtain a similar rule for case (iii), except that e_2 is type-checked instead of e_1, and t'' is tested against $\neg t$.

Finally, consider case (iv). We have to type-check both branches and each typing derivation may end with (*subsum*):

$$\dfrac{\Delta \wr \Gamma \vdash e : t' \quad \begin{cases} t' \not\leq \neg t \quad \text{and} \quad \dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1 : s_1} \quad s_1 \leq s}{\Delta \wr \Gamma \vdash e_1 : s} \\ t' \not\leq t \quad \text{and} \quad \dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e_2 : s_2} \quad s_2 \leq s}{\Delta \wr \Gamma \vdash e_2 : s} \end{cases}}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s}\ (case)$$

Subsumption is used there just to unify s_1 and s_2 into a common type s, which is used to type the whole type case. Such a common type can also be obtained by taking the least upper-bound of s_1 and s_2, i.e., $s_1 \vee s_2$. Because $s_1 \leq s$ and $s_2 \leq s$, we have $s_1 \vee s_2 \leq s$, and we can rewrite the derivation as follows:

$$\dfrac{\dfrac{\Delta \wr \Gamma \vdash e : t' \quad \begin{cases} t' \not\leq \neg t \quad \text{and} \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_1 : s_1} \\ t' \not\leq t \quad \text{and} \quad \dfrac{\cdots}{\Delta \wr \Gamma \vdash e_2 : s_2} \end{cases}}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s_1 \vee s_2}\ (case) \qquad s_1 \vee s_2 \leq s}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s}$$

Suppose now that the sub-derivation for e ends with (*subsum*):

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e : t''} \quad t'' \leq t'}{\Delta \wr \Gamma \vdash e : t'} \quad \begin{cases} t' \not\leq \neg t \quad \text{and} \quad \Delta \wr \Gamma \vdash e_1 : s_1 \\ t' \not\leq t \quad \text{and} \quad \Delta \wr \Gamma \vdash e_2 : s_2 \end{cases}}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s_1 \vee s_2}\ (case)$$

The relations $t'' \leq t'$, $t' \not\leq \neg t$ do not necessarily imply $t'' \not\leq \neg t$, and $t'' \leq t'$, $t' \not\leq t$ do not necessarily imply $t'' \not\leq \neg t$. Therefore, by using the type t'' instead of t' for e, we may type-check less branches. If so, then we would be in one of the cases $(i) - (iii)$, and the result type (*i.e.*, a type among $\mathbb{0}$, s_1 or s_2) for the whole type case would be smaller than $s_1 \vee s_2$. It would then be possible to type the type case with $s_1 \vee s_2$ by subsumption. Otherwise, we type-check as many branches with t'' as with t', and we can modify the rule into

$$\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e : t''} \quad \begin{cases} t'' \not\leq \neg t \quad \text{and} \quad \Delta \wr \Gamma \vdash e_1 : s_1 \\ t'' \not\leq t \quad \text{and} \quad \Delta \wr \Gamma \vdash e_2 : s_2 \end{cases}}{\Delta \wr \Gamma \vdash (e \in t \,?\, e_1 : e_2) : s_1 \vee s_2}$$

Finally, consider the case where the last rule in a derivation is (*instinter*) and all its sub-derivations end with (*subsum*):

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e : s} \quad s \leq t}{\Delta \wr \Gamma \vdash e : t}\ (subsum) \qquad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \wr \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\ (instinter)$$

Since $s \leq t$, we have $\bigwedge_{j\in J} s\sigma_j \leq \bigwedge_{j\in J} t\sigma_j$. So such a derivation can be rewritten into

$$\dfrac{\dfrac{\dfrac{\cdots}{\Delta \wr \Gamma \vdash e : s} \quad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \wr \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j}\ (instinter) \qquad \bigwedge_{j\in J} s\sigma_j \leq \bigwedge_{j\in J} t\sigma_j}{\Delta \wr \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\ (subsum)$$

In conclusion, by applying the aforementioned transformations repeatedly, we can rewrite an arbitrary typing derivation into a special form where subsumption are used at the end of sub-derivations of projections, abstractions or applications, in the conditions of type cases and at the very end of the whole derivation. Thus, this transformations yields a set of syntax-directed typing rules, given in Figure 6. Let $\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t$ denote the typing judgments derivable by the set of syntax-directed typing rules.

$$\frac{}{\Delta \wr \Gamma \vdash_{\mathscr{A}} c : b_c}\ (\textsc{Alg-const}) \qquad \frac{}{\Delta \wr \Gamma \vdash_{\mathscr{A}} x : \Gamma(x)}\ (\textsc{Alg-var})$$

$$\frac{\Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 : t_1 \qquad \Delta \wr \Gamma \vdash_{\mathscr{A}} e_2 : t_2}{\Delta \wr \Gamma \vdash_{\mathscr{A}} (e_1, e_2) : t_1 \times t_2}\ (\textsc{Alg-pair}) \qquad \frac{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t \qquad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \wr \Gamma \vdash_{\mathscr{A}} \pi_i(e) : \boldsymbol{\pi}_i(t)}\ (\textsc{Alg-proj})$$

$$\frac{\Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 : t \qquad \Delta \wr \Gamma \vdash_{\mathscr{A}} e_2 : s \qquad t \leq \mathbb{0} \to \mathbb{1} \qquad s \leq \mathsf{dom}(t)}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 e_2 : t \cdot s}\ (\textsc{Alg-appl})$$

$$\frac{\begin{array}{l}\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J} (t_i\sigma_j \to s_i\sigma_j)) \\ \forall i \in I, j \in J.\ \Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash_{\mathscr{A}} e@[\sigma_j] : s'_{ij} \qquad s'_{ij} \leq s_i\sigma_j\end{array}}{\Delta \wr \Gamma \vdash_{\mathscr{A}} \lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J} (t_i\sigma_j \to s_i\sigma_j)}\ (\textsc{Alg-abstr})$$

$$\frac{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : \mathbb{0}}{\Delta \wr \Gamma \vdash_{\mathscr{A}} (e \in t\,?\,e_1 : e_2) : \mathbb{0}}\ (\textsc{Alg-case-none})$$

$$\frac{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t' \qquad t' \leq t \qquad t' \not\leq \neg t \qquad \Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 : s_1}{\Delta \wr \Gamma \vdash_{\mathscr{A}} (e \in t\,?\,e_1 : e_2) : s_1}\ (\textsc{Alg-case-fst})$$

$$\frac{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t' \qquad t' \leq \neg t \qquad t' \not\leq t \qquad \Delta \wr \Gamma \vdash_{\mathscr{A}} e_2 : s_2}{\Delta \wr \Gamma \vdash_{\mathscr{A}} (e \in t\,?\,e_1 : e_2) : s_2}\ (\textsc{Alg-case-snd})$$

$$\frac{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t' \qquad \begin{cases} t' \not\leq \neg t & \text{and} \quad \Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 : s_1 \\ t' \not\leq t & \text{and} \quad \Delta \wr \Gamma \vdash_{\mathscr{A}} e_2 : s_2 \end{cases}}{\Delta \wr \Gamma \vdash_{\mathscr{A}} (e \in t\,?\,e_1 : e_2) : s_1 \vee s_2}\ (\textsc{Alg-case-both})$$

$$\frac{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t \qquad \forall j \in J.\ \sigma_j \sharp \Delta \qquad |J| > 0}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}\ (\textsc{Alg-inst})$$

Figure 6. Syntax-directed typing rules

Theorem C.22 (Soundness). *Let e be an expression. If $\Gamma \vdash_{\mathscr{A}} e : t$, then $\Gamma \vdash e : t$.*

Proof. By induction on the typing derivation of $\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t$. We proceed by a case analysis on the last rule used in the derivation.

(ALG-CONST)**:** straightforward.

(ALG-VAR): straightforward.

(ALG-PAIR): consider the derivation

$$\frac{\dots \quad \dots}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 : t_1 \qquad \Delta \wr \Gamma \vdash_{\mathscr{A}} e_2 : t_2}{\Delta \wr \Gamma \vdash_{\mathscr{A}} (e_1, e_2) : t_1 \times t_2}$$

Applying the induction hypothesis twice, we get $\Delta \wr \Gamma \vdash e_i : t_i$. Then by applying the rule (*pair*), we have $\Delta \wr \Gamma \vdash (e_1, e_2) : t_1 \times t_2$.

(ALG-PROJ): consider the derivation

$$\frac{\dots}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t \qquad t \leq \mathbb{1} \times \mathbb{1}}{\Delta \wr \Gamma \vdash_{\mathscr{A}} \pi_i(e) : \boldsymbol{\pi}_i(t)}$$

By induction, we have $\Delta \wr \Gamma \vdash e : t$. According to Lemma C.5, we have $t \leq (\boldsymbol{\pi}_1(t) \times \boldsymbol{\pi}_2(t))$. Then by (*subsum*), we get $\Delta \wr \Gamma \vdash e : (\boldsymbol{\pi}_1(t) \times \boldsymbol{\pi}_2(t))$. Finally, the rule (*proj*) gives us $\Delta \wr \Gamma \vdash \pi_i(e) : \boldsymbol{\pi}_i(t)$.

(ALG-APPL): consider the derivation

$$\frac{\dots \quad \dots \qquad t \leq \mathbb{0} \to \mathbb{1} \quad s \leq \mathsf{dom}(t)}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 : t \qquad \Delta \wr \Gamma \vdash_{\mathscr{A}} e_2 : s \qquad \Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 e_2 : t \cdot s}$$

By induction, we have $\Delta \wr \Gamma \vdash e_1 : t$ and $\Delta \wr \Gamma \vdash e_2 : s$. According to Lemma C.13, we have

$$t \cdot s = \min\{s' \mid t \leq s \to s'\}$$

Note that the conditions $t \leq \mathbb{0} \to \mathbb{1}$ and $s \leq \mathsf{dom}(t)$ ensure that such a type exists. It is clear $t \leq s \to (t \cdot s)$. Then by (*subsum*), we get $\Delta \wr \Gamma \vdash e_1 : s \to (t \cdot s)$. Finally, the rule (*appl*) gives us $\Delta \wr \Gamma \vdash e_1 e_2 : t \cdot s$.

(ALG-ABSTR): consider the derivation

$$\frac{\forall i \in I, j \in J.\ \dfrac{\dots}{\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash_{\mathscr{A}} e@[\sigma_j] : s'_{ij}} \qquad s'_{ij} \leq s_i\sigma_j}{\Delta \wr \Gamma \vdash_{\mathscr{A}} \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)}$$

with $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j))$. By induction, for all $i \in I$ and $j \in J$, we have $\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s'_{ij}$. Since $s'_{ij} \leq s_i\sigma_j$, by (*subsum*), we get $\Delta' \wr \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j$. Finally, the rule (*abstr*) gives us $\Delta \wr \Gamma \vdash \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)$.

(ALG-CASE-NONE): consider the derivation

$$\frac{\dots}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : \mathbb{0}}{\Delta \wr \Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : \mathbb{0}}$$

By induction, we have $\Delta \wr \Gamma \vdash e : \mathbb{0}$. No branch is type-checked by the rule (*case*), so any type can be assigned to the type case expression, and in particular we have $\Delta \wr \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : \mathbb{0}$

(ALG-CASE-FST): consider the derivation

$$\frac{\dots \qquad t' \leq t \quad \dfrac{\dots}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 : s_1}}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t' \qquad \Delta \wr \Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : s_1}$$

By induction, we have $\Delta \wr \Gamma \vdash e : t'$ and $\Delta \wr \Gamma \vdash e_1 : s_1$. As $t' \leq t$, then we only need to type-check the first branch. Therefore, by the rule (*case*), we have $\Delta \wr \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s_1$.

(ALG-CASE-SND): similar the case of (ALG-CASE-FST).

(ALG-CASE-BOTH): consider the derivation

$$\frac{\dots}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t'} \quad \begin{cases} t' \not\leq \neg t \quad \text{and} \quad \dfrac{\dots}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e_1 : s_1} \\ t' \not\leq t \quad \text{and} \quad \dfrac{\dots}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e_2 : s_2} \end{cases}}{\Delta \wr \Gamma \vdash_{\mathscr{A}} (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2}$$

By induction, we have $\Delta \wr \Gamma \vdash e : t'$, $\Delta \wr \Gamma \vdash e_1 : s_1$ and $\Delta \wr \Gamma \vdash e_2 : s_2$. It is clear that $s_1 \leq s_1 \vee s_2$ and $s_2 \leq s_1 \vee s_2$. Then by (*subsum*), we get $\Delta \wr \Gamma \vdash e_1 : s_1 \vee s_2$ and $\Delta \wr \Gamma \vdash e_2 : s_1 \vee s_2$. Moreover, as $t' \not\leq \neg t$ and $t' \not\leq t$, we have to type-check both branches. Finally, by the rule (*case*), we get $\Delta \wr \Gamma \vdash (e{\in}t\,?\,e_1 : e_2) : s_1 \vee s_2$.

(ALG-INST): consider the derivation

$$\frac{\dots}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e : t \qquad \forall j \in J.\ \sigma_j \sharp \Delta}{\Delta \wr \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j}$$

By induction, we have $\Delta \wr \Gamma \vdash e : t$. As $\forall j \in J.\ \sigma_j \sharp \Delta$, by (*instinter*), we get $\Delta \wr \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j$.

□

Theorem C.23 (**Completeness**). *Let $\leq$ be a subtyping relation induced by a well-founded (convex) model with infinite support and e an expression. If $\Delta \fatsemi \Gamma \vdash e : t$, then there exists a type s such that $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e : s$ and $s \leq t$.*

Proof. By induction on the typing derivation of $\Delta \fatsemi \Gamma \vdash e : t$. We proceed by case analysis on the last rule used in the derivation.

(*const*)**:** straightforward (take s as b_c).
(*var*)**:** straightforward (take s as $\Gamma(x)$).
(*pair*)**:** consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t_2}}{\Delta \fatsemi \Gamma \vdash (e_1, e_2) : t_1 \times t_2}\ (pair)$$

Applying the induction hypothesis twice, we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_i : s_i$ where $s_i \leq t_i$. Then the rule (ALG-PAIR) gives us $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} (e_1, e_2) : s_1 \times s_2$. Since $s_i \leq t_i$, we deduce $(s_1 \times s_2) \leq (t_1 \times t_2)$.
(*proj*)**:** consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : (t_1 \times t_2)}}{\Delta \fatsemi \Gamma \vdash \pi_i(e) : t_i}\ (proj)$$

By induction, there exists s such that $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e : s$ and $s \leq (t_1 \times t_2)$. Clearly we have $s \leq \mathbb{1} \times \mathbb{1}$. Applying (ALG-PROJ), we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} \pi_i(e) : \boldsymbol{\pi}_i(s)$. Moreover, as $s \leq (t_1 \times t_2)$, according to Lemma C.5, we have $\boldsymbol{\pi}_i(s) \leq t_i$. Therefore, the result follows.
(*appl*)**:** consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : t_1 \to t_2} \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : t_1}}{\Delta \fatsemi \Gamma \vdash e_1 e_2 : t_2}\ (appl)$$

Applying the induction hypothesis twice, we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_1 : t$ and $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_2 : s$ where $t \leq t_1 \to t_2$ and $s \leq t_1$. Clearly we have $t \leq \mathbb{0} \to \mathbb{1}$ and $t \leq s \to t_2$ (by contravariance of arrows). From Lemma C.12, we get $s \leq \mathsf{dom}(t)$. So, by applying the rule (ALG-APPL), we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_1 e_2 : t \cdot s$. Moreover, it is clear that t_2 is a solution for $t \leq s \to s'$. Consequently, it is a super type of $t \cdot s$, that is $t \cdot s \leq t_2$.
(*abstr*)**:** consider the derivation

$$\frac{\forall i \in I, j \in J.\ \dfrac{\cdots}{\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash e@[\sigma_j] : s_i\sigma_j}}{\Delta \fatsemi \Gamma \vdash \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)}\ (abstr)$$

where $\Delta' = \Delta \cup \mathsf{var}(\bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j))$. By induction, for all $i \in I$ and $j \in J$, there exists s'_{ij} such that $\Delta' \fatsemi \Gamma, (x : t_i\sigma_j) \vdash_{\mathscr{A}} e@[\sigma_j] : s'_{ij}$ and $s'_{ij} \leq s_i\sigma_j$. Then the rule (ALG-ABSTR) gives us $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} \lambda^{\bigwedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e : \bigwedge_{i\in I, j\in J}(t_i\sigma_j \to s_i\sigma_j)$
(*case*)**:** consider the derivation

$$\frac{\dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e : t'} \quad \begin{cases} t' \not\leq \neg t & \Rightarrow \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_1 : s} \\ t' \not\leq t & \Rightarrow \quad \dfrac{\cdots}{\Delta \fatsemi \Gamma \vdash e_2 : s} \end{cases}}{\Delta \fatsemi \Gamma \vdash (e \in t\,?\,e_1 : e_2) : s}\ (case)$$

By induction hypothesis on $\Delta \fatsemi \Gamma \vdash e : t'$, there exists a type t'' such that $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e : t''$ and $t'' \leq t'$. If $t'' \simeq \mathbb{0}$, by (ALG-CASE-NONE), we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} (e \in t\,?\,e_1 : e_2) : \mathbb{0}$. The result follows straightforwardly. In what follows, we assume that $t'' \not\simeq \mathbb{0}$.
Assume that $t'' \leq t$. Because $t'' \leq t'$, we have $t' \not\leq \neg t$ (otherwise, $t'' \simeq \mathbb{0}$). Therefore the first branch is type-checked, and by induction, there exists a type s_1 such that $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_1 : s_1$ and $s_1 \leq s$. Then the rule (ALG-CASE-FST) gives us $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} (e \in t\,?\,e_1 : e_2) : s_1$.
Otherwise, $t'' \not\leq t$. In this case, we have $t' \not\leq t$ (otherwise, $t'' \leq t$). Then the second branch is type-checked. By induction, there exists a type s_2 such that $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_2 : s_2$ and $s_2 \leq s$. If $t'' \leq \neg t$, then by the rule (ALG-CASE-SND), we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} (e \in t\,?\,e_1 : e_2) : s_2$. The result follows. Otherwise, we also have $t'' \not\leq \neg t$. Then we also have $t' \not\leq \neg t$ (otherwise, $t'' \leq \neg t$). So the first branch should be type-checked as well. By induction, we have $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e_1 : s_1$ where $s_1 \leq s$. By applying (ALG-CASE-BOTH), we get $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} (e \in t\,?\,e_1 : e_2) : s_1 \vee s_2$. Since $s_1 \leq s$ and $s_2 \leq s$, we deduce that $s_1 \vee s_2 \leq s$. The result follows as well.

(*instinter*): consider the derivation

$$\frac{\dots}{\Delta \fatsemi \Gamma \vdash e : t} \quad \forall j \in J.\, \sigma_j \not\sharp \Delta \;\; (instinter)$$
$$\Delta \fatsemi \Gamma \vdash e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} t\sigma_j$$

By induction, there exists a type s such that $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e : s$ and $s \leq t$. Then the rule (ALG-INST) gives us that $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e[\sigma_j]_{j\in J} : \bigwedge_{j\in J} s\sigma_j$. Since $s \leq t$, we have $\bigwedge_{j\in J} s\sigma_j \leq \bigwedge_{j\in J} t\sigma_j$. Therefore, the result follows.

□

Corollary C.24 (Minimum typing). *Let e be an expression. If $\Delta \fatsemi \Gamma \vdash_{\mathscr{A}} e : t$, then $t = \min\{s \mid \Delta \fatsemi \Gamma \vdash e : s\}$.*

Proof. Consequence of Theorems C.22 and C.23. □

To prove the termination of the type-checking algorithm, we define the *size* of an expression e as follows.

Definition C.25. *Let e be an expression. We define the* size *of e as:*

$$\begin{aligned}
\mathit{size}(c) &= 1\\
\mathit{size}(x) &= 1\\
\mathit{size}((e_1, e_2)) &= \mathit{size}(e_1) + \mathit{size}(e_2) + 1\\
\mathit{size}(\pi_i(e)) &= \mathit{size}(e) + 1\\
\mathit{size}(e_1 e_2) &= \mathit{size}(e_1) + \mathit{size}(e_2) + 1\\
\mathit{size}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e) &= \mathit{size}(e) + 1\\
\mathit{size}(e \in t\,?\,e_1 : e_2) &= \mathit{size}(e) + \mathit{size}(e_1) + \mathit{size}(e_2) + 1\\
\mathit{size}(e[\sigma_j]_{j\in J}) &= \mathit{size}(e) + 1
\end{aligned}$$

The relabeling does not enlarge the size of the expression.

Lemma C.26. *Let e be an expression and $[\sigma_j]_{j\in J}$ a set of type substitutions. Then*

$$\mathit{size}(e@[\sigma_j]_{j\in J}) \leq \mathit{size}(e).$$

Proof. By induction on the structure of e. □

Theorem C.27 (Termination). *Let e be an expression. Then the type-inference algorithm for e terminates.*

Proof. By induction on the sizes of the expressions to be checked.

(ALG-CONST): it terminates immediately.
(ALG-VAR): it terminates immediately.
(ALG-PAIR): $\text{size}(e_1) + \text{size}(e_2) < \text{size}((e_1, e_2))$.
(ALG-PROJ): $\text{size}(e') < \text{size}(\pi_1(e'))$.
(ALG-APPL): $\text{size}(e_1) + \text{size}(e_2) < \text{size}(e_1 e_2)$.
(ALG-ABSTR): by Lemma C.26, we have $\text{size}(e'@[\sigma_j]) \leq \text{size}(e')$. Then we get

$$\text{size}(e'@[\sigma_j]) < \text{size}(\lambda^{\wedge_{i\in I} t_i \to s_i}_{[\sigma_j]_{j\in J}} x.e').$$

(ALG-CASE): $\text{size}(e') + \text{size}(e_1) + \text{size}(e_2) < \text{size}(e' \in t\,?\,e_1 : e_2)$.
(ALG-INST): $\text{size}(e') < \text{size}(e'[\sigma_j]_{j\in J})$.

□

D. Evaluation

As described in Section 3, below we assume that polymorphic variables are pairwise distinct and distinct from any monomorphic variables in the expressions under consideration. In particular, when substituting an expression e for a variable x in an expression e', we assume the polymorphic type variables of e' to be distinct from the monomorphic and polymorphic type variables of e. Furthermore, we assume that there are no useless type variables in the domain of any type-substitution and no redundant type-substitutions in any set of type-substitutions (Lemmas B.9 and B.10).

D.1 Equivalence between small-step semantics and big-step semantics

In this section, we prove the equivalence between the small-step semantics in Section A.4 and the big-step semantics for the polymorphic language in Section 5.2 (extended with let-polymorphism in Section 5.4). For clarity, we use v for the values for the small-step semantics and v_p for those of the big-step semantics.

Definition D.1 (Polymorphic language)**.**

$$\begin{array}{rcll} e & ::= & c \mid x \mid \underline{x} \mid \lambda^t_{\sigma_I} x.e \mid ee \mid e \in t \ \texttt{?}\ e : e & \\ & \mid & e\sigma_I \mid \texttt{let}\ \underline{x} = e\ \texttt{in}\ e & \\ v & ::= & c \mid \lambda^t_{\sigma_I} x.e & \textit{(for the small-step semantics)} \\ v_\mathsf{p} & ::= & c \mid \langle \lambda^t_{\sigma_I} x.e, \mathscr{E}, \sigma_I \rangle & \textit{(for the big-step semantics)} \end{array}$$

Definition D.2 (Membership)**.** *The membership relation $v_\mathsf{p} \in_\mathsf{p} t$ for polymorphic values is inductively defined as follows:*

$$\begin{array}{rcl} c \in_\mathsf{p} t & \stackrel{\text{def}}{=} & b_c \leq t \\ \langle \lambda^s_{\sigma_I} x.e, \mathscr{E}, \sigma_J \rangle \in_\mathsf{p} t & \stackrel{\text{def}}{=} & s(\sigma_J \circ \sigma_I) \leq t \end{array}$$

Definition D.3. *We define a transformation function $(\!|.|\!)$ from v_p to v as follows:*

$$\begin{array}{rcl} (\!|c|\!) & \stackrel{\text{def}}{=} & c \\ (\!|\langle \lambda^s_{\sigma_I} x.e, \mathscr{E}, \sigma_J \rangle|\!) & \stackrel{\text{def}}{=} & \lambda^s_{\sigma_J \circ \sigma_I} x.e(\!|\mathscr{E}|\!) \end{array}$$

where $(\!|\mathscr{E}|\!)$ is defined as follows:

$$\begin{array}{rcl} (\!|\varnothing|\!) & \stackrel{\text{def}}{=} & \varnothing \\ (\!|\mathscr{E}, x \mapsto v_\mathsf{p}|\!) & \stackrel{\text{def}}{=} & (\!|\mathscr{E}|\!) \cup \{(\!|v_\mathsf{p}|\!)/x\} \\ (\!|\mathscr{E}, \underline{x} \mapsto v_\mathsf{p}|\!) & \stackrel{\text{def}}{=} & (\!|\mathscr{E}|\!) \cup \{(\!|v_\mathsf{p}|\!)/\underline{x}\} \end{array}$$

In the definition of $(\!|\langle \lambda^s_{\sigma_I} x.e, \mathscr{E}, \sigma_J \rangle|\!)$, to avoid unwanted captures, we assume that the polymorphic type variables of $\lambda^s_{\sigma_J \circ \sigma_I} x.e$ are distinct from the monomorphic and polymorphic type variables in $\mathscr{E}$, *ie*, $\mathsf{tv}(\mathscr{E})$ as precisely defined in Definition D.4. This is guaranteed by the assumption we impose on variables.

Definition D.4. *The sets $\mathsf{tv}(v_\mathsf{p})$ and $\mathsf{tv}(\mathscr{E})$ of type variables respectively occurring in v_p and $\mathscr{E}$ are defined as follows:*

$$\begin{array}{rcl} \mathsf{tv}(v_\mathsf{p}) & \stackrel{\text{def}}{=} & \mathsf{tv}((\!|v_\mathsf{p}|\!)) \\ \mathsf{tv}(\mathscr{E}) & \stackrel{\text{def}}{=} & \bigcup_{x \in \mathsf{dom}(\mathscr{E})} \mathsf{tv}(\mathscr{E}(x)) \ \cup \ \bigcup_{\underline{x} \in \mathsf{dom}(\mathscr{E})} \mathsf{tv}(\mathscr{E}(\underline{x})) \end{array}$$

Definition D.5. *Let e be an expression and $\mathscr{E}$ an environment. We write $e \sharp \mathscr{E}$ to mean that the polymorphic type variables in e are distinct from $\mathsf{tv}(\mathscr{E})$. Moreover, we define $\sharp v_\mathsf{p}$ and $\sharp\mathscr{E}$ recursively as follows:*

$$\begin{array}{rcl} \sharp\langle \lambda^t_{\sigma_I} x.e, \mathscr{E}, \sigma_J \rangle & \stackrel{\text{def}}{=} & \sharp\mathscr{E} \ \textit{and} \ \exists t', \ \vdash (\!|\langle \lambda^t_{\sigma_I} x.e, \mathscr{E}, \sigma_J \rangle|\!) : t' \ \textit{and} \ (\lambda^t_{\sigma_J \circ \sigma_I} x.e) \sharp \mathscr{E} \\ \sharp c & \stackrel{\text{def}}{=} & \texttt{true} \\ \sharp\mathscr{E} & \stackrel{\text{def}}{=} & \forall x \in \mathsf{dom}(\mathscr{E}), \ \sharp(\mathscr{E}(x)) \ \textit{and} \ \forall \underline{x} \in \mathsf{dom}(\mathscr{E}), \ \sharp(\mathscr{E}(\underline{x})) \end{array}$$

Definition D.6. *Let $\mathbb{i}$ denote a singleton set made of the empty type-substitution $[\{\,\}]$, which is the neutral element of the composition of sets of type-substitutions. We define a transformation function $\mathsf{clos}(\cdot)$ from $(v, \mathscr{E}, \sigma_I)$ to v_p as follows:*

$$\begin{array}{rcl} \mathsf{clos}(c, \mathscr{E}, \sigma_I) & \stackrel{\text{def}}{=} & c \\ \mathsf{clos}(\lambda^t_{\sigma_J} x.e, \mathscr{E}, \sigma_I) & \stackrel{\text{def}}{=} & \langle \lambda^t_{\sigma_J} x.e, \mathscr{E}, \sigma_I \rangle \end{array}$$

For simplicity, we write $\mathsf{clos}(v)$ for $\mathsf{clos}(v, \varnothing, \mathbb{i})$.

Definition D.7. *We define the reflexive and transitive closure $\rightsquigarrow^*$ of a single-step reduction $\rightsquigarrow$ by the following two rules:*

$$\frac{}{e \rightsquigarrow^* e} \ (\textit{refl}) \qquad \frac{e_1 \rightsquigarrow e_2 \qquad e_2 \rightsquigarrow^* e_3}{e_1 \rightsquigarrow^* e_3} \ (\textit{trans})$$

Lemma D.8. *Suppose $e_1 \rightsquigarrow^* e_1'$. Then:*
(1) $e_1 e_2 \rightsquigarrow^* e_1' e_2$.
(2) $e_0 e_1 \rightsquigarrow^* e_0 e_1'$.
(3) $e_1 \in t \ \texttt{?}\ e_2 : e_3 \rightsquigarrow^* e_1' \in t \ \texttt{?}\ e_2 : e_3$.
(4) $\texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2 \rightsquigarrow^* \texttt{let}\ \underline{x} = e_1'\ \texttt{in}\ e_2$.

Proof. By induction on a derivation of $e_1 \rightsquigarrow^* e_1'$. □

Lemma D.9. *If $e_1 \rightsquigarrow^* e_2$ and $e_2 \rightsquigarrow^* e_3$, then $e_1 \rightsquigarrow^* e_3$.*

Proof. By induction on a derivation of $e_1 \rightsquigarrow^* e_2$. □

Lemma D.10. *Let v_p be a polymorphic value. The following properties hold:*
(1) Suppose $v_\mathsf{p} \in_\mathsf{p} t$. If there exists some type t' such that $\vdash (\!|v_\mathsf{p}|\!) : t'$, then $\vdash (\!|v_\mathsf{p}|\!) : t$.
(2) $\vdash (\!|v_\mathsf{p}|\!) : t$ implies $v_\mathsf{p} \in_\mathsf{p} t$.

Proof. (1) By induction on a derivation of $\vdash (\!|v_\mathsf{p}|\!) : t'$. There are three cases to consider.

(*const*)**:** $v_\mathsf{p} = c$ and $t' = b_c$. By definition D.2, $b_c \leq t$ and by the rule (*subsum*), we have $\vdash c : t$.
(*abstr*)**:** $v_\mathsf{p} = \langle \lambda^s_{\sigma_I} x.e, \mathscr{E}, \sigma_J \rangle$, $(\!|v_\mathsf{p}|\!) = \lambda^s_{\sigma_J \circ \sigma_I} x.e(\!|\mathscr{E}|\!)$, and $t' = s(\sigma_J \circ \sigma_I)$.
By Definition D.2, $s(\sigma_J \circ \sigma_I) \leq t$ and by the rule (*subsum*), we have $\vdash (\!|v_\mathsf{p}|\!) : t$.
(*subsum*)**:** We have as assumptions $\vdash (\!|v_\mathsf{p}|\!) : s$ and $s \leq t'$. By induction hypothesis, we have $\vdash (\!|v_\mathsf{p}|\!) : t$.

(2) By induction on a derivation of $\vdash (\!|v_\mathsf{p}|\!) : t$. □

Lemma D.11. *Let v_p be a polymorphic value.*
(1) Suppose $v_\mathsf{p} \notin_\mathsf{p} t$. If there exists some type t' such that $\vdash (\!|v_\mathsf{p}|\!) : t'$, then $\vdash (\!|v_\mathsf{p}|\!) : \neg t$.
(2) $\vdash (\!|v_\mathsf{p}|\!) : \neg t$ implies $v_\mathsf{p} \notin_\mathsf{p} t$.

Proof. (1) By induction on a derivation of $\vdash (\!|v_\mathsf{p}|\!) : t'$. (2) The proof is by contradiction. Suppose $v_\mathsf{p} \in_\mathsf{p} t$. Then, by Lemma D.10, we have $\vdash (\!|v_\mathsf{p}|\!) : t$, which is absurd. □

Lemma D.12. *Suppose (1) $e \sharp \mathscr{E}$ and $\sharp \mathscr{E}$; (2) $\vdash (e@\sigma_I)(\!|\mathscr{E}|\!) : t$; and (3) $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e \Downarrow v_\mathsf{p}$. Then (4) $\sharp v_\mathsf{p}$ and (5) $(e@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_\mathsf{p}|\!)$.*

Proof. By induction on a derivation of $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e \Downarrow v_\mathsf{p}$.

(PE-Const)**:** $\sigma_I; \mathscr{E} \vdash_\mathsf{p} c \Downarrow c$ where $e = c$.
- $\sharp c$ and $(c@\sigma_I)(\!|\mathscr{E}|\!) = c \rightsquigarrow^* c$.

(PE-Var)**:** $\sigma_I; \mathscr{E} \vdash_\mathsf{p} x \Downarrow \mathscr{E}(x)$ where $e = x$.
- From (1), we have $\sharp(\mathscr{E}(x))$.
- By Definition D.3, $(x@\sigma_I)(\!|\mathscr{E}|\!) = x(\!|\mathscr{E}|\!) = x\{(\!|\mathscr{E}(x)|\!)/x\} = (\!|\mathscr{E}(x)|\!) \rightsquigarrow^* (\!|\mathscr{E}(x)|\!)$.

(PE-Closure)**:** $\sigma_I; \mathscr{E} \vdash_\mathsf{p} \lambda^s_{\sigma_J} x.e_0 \Downarrow \langle \lambda^s_{\sigma_J} x.e_0, \mathscr{E}, \sigma_I \rangle$ where $e = \lambda^s_{\sigma_J} x.e_0$.
- By Definition D.3, $((\lambda^s_{\sigma_J} x.e_0)@\sigma_I)(\!|\mathscr{E}|\!) = \lambda^s_{\sigma_I \circ \sigma_J} x.e_0(\!|\mathscr{E}|\!) = (\!|\langle \lambda^s_{\sigma_J} x.e_0, \mathscr{E}, \sigma_I \rangle|\!) \rightsquigarrow^* (\!|\langle \lambda^s_{\sigma_J} x.e_0, \mathscr{E}, \sigma_I \rangle|\!)$.
- Moreover, from (1) and (2), we have $\sharp \langle \lambda^s_{\sigma_J} x.e_0, \mathscr{E}, \sigma_I \rangle$.

(PE-Apply)**:** $e = e_1 e_2$.

$$\frac{(a)\ \ \sigma_I; \mathscr{E} \vdash_\mathsf{p} e_1 \Downarrow \langle \lambda^s_{\sigma_K} x.e_0, \mathscr{E}', \sigma_H \rangle \quad s = \wedge_{l \in L} s_l \to s'_l \qquad (b)\ \ \sigma_I; \mathscr{E} \vdash_\mathsf{p} e_2 \Downarrow v_{\mathsf{p}0} \quad \sigma_J = \sigma_H \circ \sigma_K \quad (c)\ \ P = \{j \in J \mid \exists l \in L : v_{\mathsf{p}0} \in_\mathsf{p} s_l \sigma_j\} \qquad (d)\ \ \sigma_P; \mathscr{E}', x \mapsto v_{\mathsf{p}0} \vdash_\mathsf{p} e_0 \Downarrow v_\mathsf{p}}{\sigma_I; \mathscr{E} \vdash_\mathsf{p} e_1 e_2 \Downarrow v_\mathsf{p}}$$

- From (2), $(e_1@\sigma_I)(\!|\mathscr{E}|\!)$ and $(e_2@\sigma_I)(\!|\mathscr{E}|\!)$ are also well-typed.
- By IH on (a), $\sharp \langle \lambda^s_{\sigma_K} x.e_0, \mathscr{E}', \sigma_H \rangle$ and $(e_1@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|\langle \lambda^s_{\sigma_K} x.e_0, \mathscr{E}', \sigma_H \rangle|\!) = \lambda^s_{\sigma_H \circ \sigma_K} x.e_0(\!|\mathscr{E}'|\!) = \lambda^s_{\sigma_J} x.e_0(\!|\mathscr{E}'|\!)$ where $\mathsf{dom}(\sigma_J) \cap \mathsf{tv}(\mathscr{E}') = \emptyset$.
- By IH on (b), $\sharp v_{\mathsf{p}0}$ and $(e_2@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_{\mathsf{p}0}|\!)$.
- By Lemmas D.8 and D.9, $((e_1 e_2)@\sigma_I)(\!|\mathscr{E}|\!) = ((e_1@\sigma_I)(\!|\mathscr{E}|\!))((e_2@\sigma_I)(\!|\mathscr{E}|\!)) \rightsquigarrow^* (\lambda^s_{\sigma_J} x.e_0(\!|\mathscr{E}'|\!))(\!|v_{\mathsf{p}0}|\!) \rightsquigarrow ((e_0(\!|\mathscr{E}'|\!))@\sigma_{P'})\{(\!|v_{\mathsf{p}0}|\!)/x\}$ where
$$(e) \quad P' = \{j \in J \mid \exists l \in L, \vdash (\!|v_{\mathsf{p}0}|\!) : s_l \sigma_j\}$$
Furthermore, by Theorem B.15 with (2), $((e_0(\!|\mathscr{E}'|\!))@\sigma_{P'})\{(\!|v_{\mathsf{p}0}|\!)/x\}$ is also well-typed.
- By $\sharp \mathscr{E}'$ and $\sharp v_{\mathsf{p}0}$, we have $\sharp(\mathscr{E}' x \mapsto v_{\mathsf{p}0})$ and by assuming α-conversion, we have $e_0 \sharp (\mathscr{E}' x \mapsto v_{\mathsf{p}0})$.
- By Lemma D.10 with (c) and (e), $P = P'$. Moreover, by Lemma B.5 with $\mathsf{dom}(\sigma_J) \cap \mathsf{tv}(\mathscr{E}') = \emptyset$, we have $(e_0(\!|\mathscr{E}'|\!))@\sigma_{P'} = (e_0(\!|\mathscr{E}'|\!))@\sigma_P = (e_0@\sigma_P)(\!|\mathscr{E}'|\!)$.
- By IH on (d), $(e_0@\sigma_P)(\!|\mathscr{E}', x \mapsto v_{\mathsf{p}0}|\!) = (e_0@\sigma_P)(\!|\mathscr{E}'|\!)\{(\!|v_{\mathsf{p}0}|\!)/x\} = ((e_0(\!|\mathscr{E}'|\!))@\sigma_P)\{(\!|v_{\mathsf{p}0}|\!)/x\} \rightsquigarrow^* (\!|v_\mathsf{p}|\!)$ and $\sharp\, v_\mathsf{p}$.
- Finally, by Lemma D.9, $((e_1 e_2)@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_\mathsf{p}|\!)$.

(PE-Type Case T)**:** $e = e_1 \in s\ ?\ e_2 : e_3$.

$$\frac{(a)\ \ \sigma_I; \mathscr{E} \vdash_\mathsf{p} e_1 \Downarrow v_{\mathsf{p}0} \qquad (b)\ \ v_{\mathsf{p}0} \in_\mathsf{p} s \qquad (c)\ \ \sigma_I; \mathscr{E} \vdash_\mathsf{p} e_2 \Downarrow v_\mathsf{p}}{\sigma_I; \mathscr{E} \vdash_\mathsf{p} e_1 \in s\ ?\ e_2 : e_3 \Downarrow v_\mathsf{p}}$$

- By definition, $((e_1 \in s\ ?\ e_2 : e_3)@\sigma_I)(\!|\mathscr{E}|\!) = (e_1@\sigma_I)(\!|\mathscr{E}|\!) \in s\ ?\ (e_2@\sigma_I)(\!|\mathscr{E}|\!) : (e_3@\sigma_I)(\!|\mathscr{E}|\!)$, and each sub-expression $(e_i@\sigma_I)(\!|\mathscr{E}|\!)$ $(i = 1, 2, 3)$ is well-typed from the assumption (2).
- By IH on (a), $(e_1@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_{\mathsf{p}0}|\!)$ and $\sharp v_{\mathsf{p}0}$.
- Then, by Lemma D.10 with (b), we have $\vdash (\!|v_{\mathsf{p}0}|\!) : s$.
- By IH on (c), $(e_2@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_\mathsf{p}|\!)$ and $\sharp v_\mathsf{p}$.
- Finally, by Lemmas D.8 and D.9, we have $(e_1@\sigma_I)(\!|\mathscr{E}|\!) \in s\ ?\ (e_2@\sigma_I)(\!|\mathscr{E}|\!) : (e_3@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_{\mathsf{p}0}|\!) \in s\ ?\ (e_2@\sigma_I)(\!|\mathscr{E}|\!) : (e_3@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow (e_2@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_\mathsf{p}|\!)$, thus proving (5).

(PE-TYPE CASE F): Similar to the case for the rule (PE-TYPE CASE T), except that we exploit Lemma D.11 instead of Lemma D.10.

(PE-INSTANCE): $e = e_0\sigma_J$.

$$\frac{(a)\ \ \sigma_I \circ \sigma_J; \mathscr{E} \vdash_{\mathsf{p}} e_0 \Downarrow v}{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_0\sigma_J \Downarrow v}$$

- By the assumption (1), we have $e_0 \sharp \mathscr{E}$.
- By IH on (a), $(e_0@(\sigma_I \circ \sigma_J))(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_{\mathsf{p}}|\!)$ and $\sharp v_{\mathsf{p}}$.
- Finally, by definition, we have $((e\sigma_J)@\sigma_I)(\!|\mathscr{E}|\!) = (e@(\sigma_I \circ \sigma_J))(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_{\mathsf{p}}|\!)$, thus proving (5).

(PE-PVAR$_c$): $e = \underline{x}$.

$$\frac{\mathscr{E}(\underline{x}) = c}{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} \underline{x} \Downarrow c}$$

- $\sharp c$ and $(\underline{x}@\sigma_I)(\!|\mathscr{E}|\!) = c \rightsquigarrow^* c$.

(PE-PVAR$_f$): $e = \underline{x}$.

$$\frac{\mathscr{E}(\underline{x}) = \langle \lambda^s_{\sigma_K} y.e_0, \mathscr{E}', \sigma_J \rangle}{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} \underline{x} \Downarrow \langle \lambda^s_{\sigma_K} y.e_0, \mathscr{E}', \sigma_I \circ \sigma_J \rangle}$$

- $(\underline{x}@\sigma_I)(\!|\mathscr{E}|\!) = (\underline{x}[\sigma_i]_{i\in I})(\!|\mathscr{E}|\!) = (\!|\mathscr{E}(\underline{x})|\!)[\sigma_i]_{i\in I} = (\!|\langle \lambda^s_{\sigma_K} y.e_0, \mathscr{E}', \sigma_J \rangle|\!)[\sigma_i]_{i\in I} \rightsquigarrow^* \lambda^s_{\sigma_I \circ (\sigma_J \circ \sigma_K)} y.e_0(\!|\mathscr{E}'|\!) = (\!|\langle \lambda^s_{\sigma_K} y.e_0, \mathscr{E}', \sigma_I \circ \sigma_J \rangle|\!)$.

(PE-LET): $e = \texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2$.

$$\frac{(a)\ \ \sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_1 \Downarrow v_{\mathsf{p}0} \qquad (b)\ \ \sigma_I; \mathscr{E}, \underline{x} \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}}{\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} \texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2 \Downarrow v_{\mathsf{p}}}$$

- By IH on (a), $(e_1@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* (\!|v_{\mathsf{p}0}|\!)$.
- By IH on (b), $(e_2@\sigma_I)(\!|\mathscr{E}, \underline{x} \mapsto v_{\mathsf{p}0}|\!) = (e_2@\sigma_I)(\!|\mathscr{E}|\!)\{(\!|v_{\mathsf{p}0}|\!)/\underline{x}\} \rightsquigarrow^* (\!|v_{\mathsf{p}}|\!)$.
- By Lemmas D.8 and D.9, $((\texttt{let}\ \underline{x} = e_1\ \texttt{in}\ e_2)@\sigma_I)(\!|\mathscr{E}|\!) = \texttt{let}\ \underline{x} = (e_1@\sigma_I)(\!|\mathscr{E}|\!)\ \texttt{in}\ (e_2@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow^* \texttt{let}\ \underline{x} = (\!|v_{\mathsf{p}0}|\!)\ \texttt{in}\ (e_2@\sigma_I)(\!|\mathscr{E}|\!) \rightsquigarrow (e_2@\sigma_I)(\!|\mathscr{E}|\!)\{(\!|v_{\mathsf{p}0}|\!)/\underline{x}\} \rightsquigarrow^* v_{\mathsf{p}}$.

□

Definition D.13. *$v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$ if and only if $(\!|v_{\mathsf{p}}|\!) = (\!|v_{\mathsf{p}}'|\!)$.*

Lemma D.14. *Suppose $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$. Then $v_{\mathsf{p}} \in_{\mathsf{p}} t$ if and only if $v_{\mathsf{p}}' \in_{\mathsf{p}} t$.*

Proof. Suppose $v_{\mathsf{p}} = c$. Then $v_{\mathsf{p}}' = c$, which completes the proof. Now suppose $v_{\mathsf{p}} = \langle \lambda^s_{\sigma_I} x.e, \mathscr{E}, \sigma_J \rangle$. Then $v_{\mathsf{p}}' = \langle \lambda^s_{\sigma_K} x.e', \mathscr{E}', \sigma_H \rangle$ where $\sigma_J \circ \sigma_I = \sigma_H \circ \sigma_K$ and $e(\!|\mathscr{E}|\!) = e'(\!|\mathscr{E}'|\!)$. By the definition of $\in_{\mathsf{p}}$, we complete the proof. □

Lemma D.15. *Suppose (1) $(e_1@\sigma_I)(\!|\mathscr{E}_1|\!) = (e_2@\sigma_I)(\!|\mathscr{E}_2|\!)$ and (2) $\sigma_I; \mathscr{E}_1 \vdash_{\mathsf{p}} e_1 \Downarrow v_{\mathsf{p}}$. Then, there exists v_{p}' such that $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.*

Proof. By induction on a derivation of $\sigma_I; \mathscr{E}_1 \vdash_{\mathsf{p}} e_1 \Downarrow v_{\mathsf{p}}$.

(PE-CONST): $e_1 = v_{\mathsf{p}} = c$.

- From (1), e_2 is either c, x, or $\underline{x}$. If $e_2 = c$, then the proof is easy.
- If $e_2 = x$, then $v_{\mathsf{p}}' = \mathscr{E}_2(x)$. From (1) we have $(e_2@\sigma_I)(\!|\mathscr{E}_2|\!) = (x@\sigma_I)(\!|\mathscr{E}_2|\!) = (\!|\mathscr{E}_2(x)|\!) = c$, thus completing the proof. The proof for $e_2 = \underline{x}$ is similar.

(PE-VAR): $e_1 = x$ and $v_{\mathsf{p}} = \mathscr{E}_1(x)$.

- From (1), e_2 is either y, c, $\lambda^s_{\sigma_K} z.e$, or $\underline{x}$.
- If $e_2 = c$, then $v_{\mathsf{p}}' = c$. Moreover, from (1) we have $(e_1@\sigma_I)(\!|\mathscr{E}_1|\!) = (x@\sigma_I)(\!|\mathscr{E}_1|\!) = (\!|\mathscr{E}_1(x)|\!) = c$, thus completing the proof.
- If $e_2 = y$, then $v_{\mathsf{p}}' = \mathscr{E}_2(y)$. From (1) we have $\mathscr{E}_1(x) = \mathscr{E}_2(y)$, thus completing the proof.
- If $e_2 = \lambda^s_{\sigma_K} z.e$, then $v_{\mathsf{p}}' = \langle e_2, \mathscr{E}_2, \sigma_I \rangle$. From (1) we have $\mathscr{E}_1(x) = (e_2@\sigma_I)(\!|\mathscr{E}_2|\!) = (\!|v_{\mathsf{p}}'|\!)$, thus completing the proof.
- Now assume $e_2 = \underline{x}$. If $\mathscr{E}_2(\underline{x}) = c$, then from (1) we also have $\mathscr{E}_1(x) = c$, thus completing the proof. Otherwise, $\mathscr{E}_2(\underline{x}) = \langle \lambda^s_{\sigma_K} z.e, \mathscr{E}', \sigma_H \rangle$ and $v_{\mathsf{p}}' = \langle \lambda^s_{\sigma_K} z.e, \mathscr{E}', \sigma_I \circ \sigma_H \rangle$. From (1), we have $\mathscr{E}_1(x) = (\underline{x}@\sigma_I)(\!|\mathscr{E}_2|\!) = (\!|\mathscr{E}_2(\underline{x})|\!)@\sigma_I = (\!|v_{\mathsf{p}}'|\!)$, thus completing the proof.

(PE-CLOSURE): $e_1 = \lambda^s_{\sigma_K} z.e$ and $v_{\mathsf{p}} = \langle e_1, \mathscr{E}_1, \sigma_I \rangle$.

- From (1), e_2 is either x, $\lambda^s_{\sigma_J} z.e'$, or $\underline{x}$.
- If $e_2 = x$, then the proof is similar to the third case for (PE-VAR).
- If $e_2 = \lambda^s_{\sigma_J} z.e'$, then $v_{\mathsf{p}}' = \langle \lambda^s_{\sigma_J} z.e', \mathscr{E}_2, \sigma_I \rangle$. From (1), we have $(\!|v_{\mathsf{p}}|\!) = (\!|v_{\mathsf{p}}'|\!)$, thus completing the proof.
- If $e_2 = \underline{x}$, then $\mathscr{E}_2(\underline{x}) = \langle \lambda^s_{\sigma_J} z.e', \mathscr{E}', \sigma_H \rangle$ and $v_{\mathsf{p}}' = \langle \lambda^s_{\sigma_J} z.e', \mathscr{E}', \sigma_I \circ \sigma_H \rangle$. From (1), we have $(\!|v_{\mathsf{p}}|\!) = (e_1@\sigma_I)(\!|\mathscr{E}_1|\!) = (\underline{x}@\sigma_I)(\!|\mathscr{E}_2|\!) = (\!|\mathscr{E}_2(\underline{x})|\!)@\sigma_I = (\!|v_{\mathsf{p}}'|\!)$, thus completing the proof.

(PE-APPLY): $e_1 = e_{11}e_{12}$.

- From (1), we have $e_2 = e_{21}e_{22}$ and $(e_{1l}@\sigma_I)[\![\mathscr{E}_1]\!] = (e_{2l}@\sigma_I)[\![\mathscr{E}_2]\!]$ where $l \in \{1,2\}$.
- From (2), we have the following assumptions:
  (a) $\sigma_I; \mathscr{E}_1 \vdash_{\mathsf{p}} e_{11} \Downarrow \langle \lambda^s_{\sigma_K} x.e, \mathscr{E}, \sigma_H \rangle$ where $s = \wedge_{i\in I} s_i \to s'_i$ and $\mathsf{tv}(\mathscr{E}) \cap \mathsf{dom}(\sigma_H \circ \sigma_K) = \emptyset$;
  (b) $\sigma_I; \mathscr{E}_1 \vdash_{\mathsf{p}} e_{12} \Downarrow v_{\mathsf{p}0}$;
  (c) $\sigma_J = \sigma_H \circ \sigma_K$ and $P = \{j \in J \mid \exists i \in I : v_{\mathsf{p}0} \in_{\mathsf{p}} s_i\sigma_j\}$; and
  (d) $\sigma_P; \mathscr{E}, x \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}$.
- By IH on (a), $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_{21} \Downarrow \langle \lambda^s_{\sigma_{K'}} x.e', \mathscr{E}', \sigma_{H'} \rangle$ where $\sigma_H \circ \sigma_K = \sigma_{H'} \circ \sigma_{K'}$ and $e[\![\mathscr{E}]\!] = e'[\![\mathscr{E}']\!]$. Moreover, $\mathsf{tv}(\mathscr{E}') \cap \mathsf{dom}(\sigma_{H'} \circ \sigma_{K'}) = \emptyset$.
- By IH on (b), $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_{22} \Downarrow v_{\mathsf{p}0}'$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$.
- By Lemma D.14 with $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$ and $\sigma_J = \sigma_H \circ \sigma_K = \sigma_{H'} \circ \sigma_{K'}$, we have $P = \{j \in J \mid \exists i \in I : v_{\mathsf{p}0} \in_{\mathsf{p}} s_i\sigma_j\} = \{j \in J \mid \exists i \in I : v_{\mathsf{p}0}' \in_{\mathsf{p}} s_i\sigma_j\}$.
- From $e[\![\mathscr{E}]\!] = e'[\![\mathscr{E}']\!]$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$ and $\mathsf{tv}(\mathscr{E}) \cap \sigma_P = \emptyset$ and $\mathsf{tv}(\mathscr{E}') \cap \sigma_P = \emptyset$, we have $(e@\sigma_P)[\![\mathscr{E}, x \mapsto v_{\mathsf{p}0}]\!] = (e'@\sigma_P)[\![\mathscr{E}', x \mapsto v_{\mathsf{p}0}']\!]$.
- Now, by IH on (d), $\sigma_P; \mathscr{E}', x \mapsto v_{\mathsf{p}0}' \vdash_{\mathsf{p}} e' \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.
- Finally, we have $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}'$ by the rule (PE-APPLY), thus completing the proof.

(PE-TYPE CASE T): $e_1 = e_{11} \in s \,?\, e_{12} : e_{13}$.
- From (1), we have $e_2 = e_{21} \in s \,?\, e_{22} : e_{23}$ and $(e_{1l}@\sigma_I)[\![\mathscr{E}_1]\!] = (e_{2l}@\sigma_I)[\![\mathscr{E}_2]\!]$ where $l \in \{1,2,3\}$.
- From (2), we have the following assumptions:
  (a) $\sigma_I; \mathscr{E}_1 \vdash_{\mathsf{p}} e_{11} \Downarrow v_{\mathsf{p}0}$ (b) $v_{\mathsf{p}0} \in_{\mathsf{p}} s$ (c) $\sigma_I; \mathscr{E}_1 \vdash_{\mathsf{p}} e_{12} \Downarrow v_{\mathsf{p}}$
- By IH on (a), $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_{21} \Downarrow v_{\mathsf{p}0}'$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$.
- By Lemma D.14 with (b), we have $v_{\mathsf{p}0}' \in_{\mathsf{p}} s$.
- By IH on (c), $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_{22} \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.
- Finally, we have $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}'$ by the rule (PE-TYPE CASE T), thus completing the proof.

(PE-TYPE CASE F): $e_1 = e_{11} \in s \,?\, e_{12} : e_{13}$.
- Similar to the case for the rule (PE-TYPE CASE F).

(PE-INSTANCE): $e_1 = e\sigma_J$.
- From (1), we have $e_2 = e'\sigma_J$ and $(e@(\sigma_I \circ \sigma_J))[\![\mathscr{E}_1]\!] = (e'@(\sigma_I \circ \sigma_J))[\![\mathscr{E}_2]\!]$.
- From (2), we have the following assumption: (a) $\sigma_I \circ \sigma_J; \mathscr{E}_1 \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}$.
- By IH on (a), $\sigma_I \circ \sigma_J; \mathscr{E}_2 \vdash_{\mathsf{p}} e' \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.
- Finally, we have $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}'$ by the rule (PE-INSTANCE), thus completing the proof.

(PE-PVAR$_c$): $e_1 = \underline{x}$ and $v_{\mathsf{p}} = c$.
- Similar to the case for (PE-CONST).

(PE-PVAR$_f$): $e_1 = \underline{x}$, $\mathscr{E}_1(\underline{x}) = \langle \lambda^s_{\sigma_K} z.e, \mathscr{E}, \sigma_H \rangle$ and $v_{\mathsf{p}} = \langle \lambda^s_{\sigma_K} z.e, \mathscr{E}, \sigma_I \circ \sigma_H \rangle$.
- Similar to the case for (PE-VAR).

(PE-LET): $e_1 = \mathtt{let}\ \underline{x} = e_{11}\ \mathtt{in}\ e_{12}$.
- From (1), we have $e_2 = \mathtt{let}\ \underline{x} = e_{21}\ \mathtt{in}\ e_{22}$. Moreover, $(e_{11}@\sigma_I)[\![\mathscr{E}_1]\!] = (e_{21}@\sigma_I)[\![\mathscr{E}_2]\!]$ and $(e_{12}@\sigma_I)[\![\mathscr{E}_1]\!] = (e_{22}@\sigma_I)[\![\mathscr{E}_2]\!]$.
- From (2), we have the following assumptions:
  (a) $\sigma_I; \mathscr{E}_1 \vdash_{\mathsf{p}} e_{11} \Downarrow v_{\mathsf{p}0}$ (b) $\sigma_I; \mathscr{E}_1, \underline{x} \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} e_{12} \Downarrow v_{\mathsf{p}}$
- By IH on (a), $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_{21} \Downarrow v_{\mathsf{p}0}'$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$.
- By IH on (b), $\sigma_I; \mathscr{E}_2, \underline{x} \mapsto v_{\mathsf{p}0}' \vdash_{\mathsf{p}} e_{22} \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.
- We conclude $\sigma_I; \mathscr{E}_2 \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}'$ by the rule (PE-LET).

□

Lemma D.16. *Let v be a well-typed closed value such that $\mathsf{dom}(\sigma_I) \cap \mathsf{tv}(v) = \emptyset$. Suppose (1) $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} (e@\sigma_L)\{v/x\} \Downarrow v_{\mathsf{p}}$. Then $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.*

Proof. By induction on a derivation of $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} (e@\sigma_L)\{v/x\} \Downarrow v_{\mathsf{p}}$.

(PE-CONST): $e = c$ and $v_{\mathsf{p}} = v_{\mathsf{p}}' = c$.

(PE-VAR): There are two cases to consider.
- Suppose $e = x$. Then $(e@\sigma_L)\{v/x\} = v$ and $v_{\mathsf{p}} = \mathsf{clos}(v, \mathscr{E}, \sigma_I) = \mathsf{clos}(v) = v_{\mathsf{p}}'$.
- Now suppose $(e = y \ \wedge\ y \neq x)$. Then $(e@\sigma_L)\{v/x\} = y$ and $v_{\mathsf{p}} = v_{\mathsf{p}}' = \mathscr{E}(y)$.

(PE-CLOSURE): $e = \lambda^s_{\sigma_K} y.e_0$, $(e@\sigma_L)\{v/x\} = \lambda^s_{\sigma_L \circ \sigma_K} y.(e_0\{v/x\})$, and $v_{\mathsf{p}} = \langle \lambda^s_{\sigma_L \circ \sigma_K} y.(e_0\{v/x\}), \mathscr{E}, \sigma_I \rangle$. Moreover, $v_{\mathsf{p}}' = \langle e, (\mathscr{E}, x \mapsto \mathsf{clos}(v)), \sigma_I \circ \sigma_L \rangle$ and thus $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.

(PE-APPLY): $e = e_1 e_2$ and $(e@\sigma_L)\{v/x\} = ((e_1@\sigma_L)\{v/x\})((e_2@\sigma_L)\{v/x\})$.
- From (1), we have the following assumptions:
  (a) $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} (e_1@\sigma_L)\{v/x\} \Downarrow \langle \lambda^s_{\sigma_K} y.e_0, \mathscr{E}_0, \sigma_H \rangle$
  where $s = \wedge_{i\in I} s_i \to s'_i$ and $\mathsf{tv}(\mathscr{E}_0) \cap \mathsf{dom}(\sigma_H \circ \sigma_K) = \emptyset$;
  (b) $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} (e_2@\sigma_L)\{v/x\} \Downarrow v_{\mathsf{p}0}$;
  (c) $\sigma_J = \sigma_H \circ \sigma_K$ and $P = \{j \in J \mid \exists i \in I : v_{\mathsf{p}0} \in_{\mathsf{p}} s_i\sigma_j\}$; and

(d) $\sigma_P; \mathscr{E}_0, y \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} e_0 \Downarrow v_{\mathsf{p}}$.

- By IH on (a), $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_1 \Downarrow \langle \lambda^s_{\sigma_{K'}} y.e_0', \mathscr{E}_0', \sigma_{H'} \rangle$ where $\sigma_H \circ \sigma_K = \sigma_{H'} \circ \sigma_{K'}$ and $e_0[\![\mathscr{E}_0]\!] = e_0'[\![\mathscr{E}_0']\!]$. Moreover, $\mathsf{tv}(\mathscr{E}_0') \cap \mathsf{dom}(\sigma_{H'} \circ \sigma_{K'}) = \emptyset$.
- By IH on (b), $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}0}'$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$.
- By Lemma D.14 with $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$ and $\sigma_H \circ \sigma_K = \sigma_{H'} \circ \sigma_{K'}$, we have $P = \{j \in J \mid \exists i \in I : v_{\mathsf{p}0} \in_{\mathsf{p}} s_i\sigma_j\} = \{j \in J \mid \exists i \in I : v_{\mathsf{p}0}' \in_{\mathsf{p}} s_i\sigma_j\}$.
- From $e_0[\![\mathscr{E}_0]\!] = e_0'[\![\mathscr{E}_0']\!]$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$ and $\mathsf{tv}(\mathscr{E}_0) \cap \sigma_P = \emptyset$ and $\mathsf{tv}(\mathscr{E}_0') \cap \sigma_P = \emptyset$, we have $(e_0@\sigma_P)[\![\mathscr{E}_0, y \mapsto v_{\mathsf{p}0}]\!] = (e_0'@\sigma_P)[\![\mathscr{E}_0', y \mapsto v_{\mathsf{p}0}']\!]$.
- By Lemma D.15 with (d), we have $\sigma_P; \mathscr{E}_0', y \mapsto v_{\mathsf{p}0}' \vdash_{\mathsf{p}} e_0' \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.
- By the rule (PE-Apply), we have $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}'$, thus completing the proof.

(PE-Type Case T): $e = e_1 \in s \,?\, e_2 : e_3$.

- $(e@\sigma_L)\{v/x\} = (e_1@\sigma_L)\{v/x\} \in s \,?\, (e_2@\sigma_L)\{v/x\} : (e_3@\sigma_L)\{v/x\}$.
- From (1), we have the following assumptions:
  (a) $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} (e_1@\sigma_L)\{v/x\} \Downarrow v_{\mathsf{p}0}$ (b) $v_{\mathsf{p}0} \in_{\mathsf{p}} s$ (c) $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} (e_2@\sigma_L)\{v/x\} \Downarrow v_{\mathsf{p}}$
- By IH on (a), $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_1 \Downarrow v_{\mathsf{p}0}'$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$.
- By Lemma D.14 with (b) and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$, we have $v_{\mathsf{p}0}' \in_{\mathsf{p}} s$.
- By IH on (c), $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.
- We conclude $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}'$ by the rule (PE-Type Case T).

(PE-Type Case F): Similar to the case for the rule (PE-Type Case T).

(PE-Instance): $e = e_0\sigma_J$ and $e' = ((e_0\sigma_J)@\sigma_L)\{v/x\} = (e_0@(\sigma_L \circ \sigma_J))\{v/x\}$.

- From (1), we have $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} (e_0@(\sigma_L \circ \sigma_J))\{v/x\} \Downarrow v_{\mathsf{p}}$.
- By IH, we have $\sigma_I \circ (\sigma_L \circ \sigma_J); \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_0 \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.
- We conclude $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_0\sigma_J \Downarrow v_{\mathsf{p}}'$ by the rule (PE-Instance).

(PE-Var$_c$): $e = \underline{x}$ and $v_{\mathsf{p}} = v_{\mathsf{p}}' = c$.

(PE-Var$_f$): $e = \underline{x}$, $\mathscr{E}(\underline{x}) = \langle \lambda^s_{\sigma_K} y.e_0, \mathscr{E}', \sigma_H \rangle$, and $v_{\mathsf{p}} = v_{\mathsf{p}}' = \langle \lambda^s_{\sigma_K} y.e_0, \mathscr{E}', (\sigma_I \circ \sigma_L) \circ \sigma_H \rangle$.

(PE-Let): $e = \mathtt{let}\ \underline{x} = e_1\ \mathtt{in}\ e_2$.

- $(e@\sigma_L)\{v/x\} = \mathtt{let}\ \underline{x} = (e_1@\sigma_L)\{v/x\}\ \mathtt{in}\ (e_2@\sigma_L)\{v/x\}$.
- From (1), we have the following assumptions:
  (a) $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} (e_1@\sigma_L)\{v/x\} \Downarrow v_{\mathsf{p}0}$ (b) $\sigma_I; \mathscr{E}, \underline{x} \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} (e_2@\sigma_L)\{v/x\} \Downarrow v_{\mathsf{p}}$
- By IH on (a), $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_1 \Downarrow v_{\mathsf{p}0}'$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} v_{\mathsf{p}0}'$.
- By IH on (b), $\sigma_I \circ \sigma_L; \mathscr{E}, \underline{x} \mapsto v_{\mathsf{p}0}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.
- We conclude $\sigma_I \circ \sigma_L; \mathscr{E}, x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}'$ by the rule (PE-Let).

□

Lemma D.17. *Let v be a well-typed closed value such that $\mathsf{dom}(\sigma_I) \cap \mathsf{tv}(v) = \emptyset$. Suppose (1) $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e\{v/\underline{x}\} \Downarrow v_{\mathsf{p}}$. Then $\sigma_I; \mathscr{E}, \underline{x} \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.*

Proof. Similar to the proof for Lemma D.16. □

Lemma D.18. *If (1) $\vdash e : t$, (2) $e \rightsquigarrow e'$, and (3) $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e' \Downarrow v_{\mathsf{p}}$, then $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.*

Proof. By induction on a derivation of $e \rightsquigarrow e'$.

$(e = e_1e_2)$:

(I) $\dfrac{e_1 \rightsquigarrow e_1'}{e_1e_2 \rightsquigarrow e_1'e_2}$ where $e' = e_1'e_2$:

- From (3), we have the following assumptions:
  (a) $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1' \Downarrow \langle \lambda^s_{\sigma_K} x.e_0, \mathscr{E}, \sigma_H \rangle$ where $s = \wedge_{l \in L} s_l \to s_l'$ and $\mathsf{tv}(\mathscr{E}) \cap \mathsf{dom}(\sigma_H \circ \sigma_K) = \emptyset$;
  (b) $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}0}$;
  (c) $\sigma_J = \sigma_H \circ \sigma_K$ and $P = \{j \in J \mid \exists l \in L : v_{\mathsf{p}0} \in_{\mathsf{p}} s_l\sigma_j\}$; and
  (d) $\sigma_P; \mathscr{E}, x \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} e_0 \Downarrow v_{\mathsf{p}}$.
- By IH on $e_1 \rightsquigarrow e_1'$ with (a), $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow \langle \lambda^s_{\sigma_{K'}} x.e_0', \mathscr{E}', \sigma_{H'} \rangle$ where $\sigma_H \circ \sigma_K = \sigma_{H'} \circ \sigma_{K'}$ and $\mathsf{tv}(\mathscr{E}') \cap \mathsf{dom}(\sigma_{H'} \circ \sigma_{K'}) = \emptyset$ and $e_0[\![\mathscr{E}]\!] = e_0'[\![\mathscr{E}']\!]$.
- From $e_0[\![\mathscr{E}]\!] = e_0'[\![\mathscr{E}']\!]$, $\mathsf{tv}(\mathscr{E}) \cap \mathsf{dom}(\sigma_P) = \emptyset$, and $\mathsf{tv}(\mathscr{E}') \cap \mathsf{dom}(\sigma_P) = \emptyset$, we have $(e_0@\sigma_P)[\![\mathscr{E}, x \mapsto v_{\mathsf{p}0}]\!] = (e_0'@\sigma_P)[\![\mathscr{E}', x \mapsto v_{\mathsf{p}}]\!]$.
- By Lemma D.15 with (d), $\sigma_P; \mathscr{E}', x \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} e_0' \Downarrow v_{\mathsf{p}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} v_{\mathsf{p}}'$.
- Finally, by the rule (PE-Apply), we conclude $\sigma_I; \mathscr{E} \vdash_{\mathsf{p}} e_1e_2 \Downarrow v_{\mathsf{p}}'$.

(II) $\dfrac{e_2 \rightsquigarrow e_2'}{e_1e_2 \rightsquigarrow e_1e_2'}$ where $e' = e_1e_2'$:

- Similar to the case for (I).

(III) $e_1 = \lambda^{\wedge_{i \in I} s_i \to s_i'}_{\sigma_J} x.e_0$, $e_2 = v$, $e' = (e_0@\sigma_P)\{v/x\}$, and $P = \{j \in J \mid \exists i \in I, \vdash v : s_i\sigma_j\}$:

- We have $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow \mathsf{clos}(e_1)$ and $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} v \Downarrow \mathsf{clos}(v)$.
- By Lemma D.10, $P = \{j \in J \mid \exists i \in I,\ \vdash v : s_i\sigma_j\} = \{j \in J \mid \exists i \in I,\ \mathsf{clos}(v) \in_{\mathsf{p}} s_i\sigma_j\}$.
- By Lemma D.16 with (3), we have $\sigma_P; x \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_0 \Downarrow {v_{\mathsf{p}}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} {v_{\mathsf{p}}}'$.
- Finally, by the rule (PE-APPLY), we conclude $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 e_2 \Downarrow {v_{\mathsf{p}}}'$.

($e = e_1 \in s \,?\, e_2 : e_3$):

(I) $\dfrac{e_1 \rightsquigarrow e_1'}{e \rightsquigarrow e_1' \in s \,?\, e_2 : e_3}$ where $e' = e_1' \in s \,?\, e_2 : e_3$:
- From (3), there are two cases to consider. Here we consider only the case for (PE-TYPE CASE T). The case for (PE-TYPE CASE F) is similar.
- We have the following assumptions:
$$(a)\ \mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1' \Downarrow v_{\mathsf{p}0} \quad (b)\ v_{\mathsf{p}0} \in_{\mathsf{p}} s \quad (c)\ \mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}$$
- By IH on $e_1 \rightsquigarrow e_1'$ with (a), $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow {v_{\mathsf{p}0}}'$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} {v_{\mathsf{p}0}}'$.
- By Lemma D.14 with (b), we have ${v_{\mathsf{p}0}}' \in_{\mathsf{p}} s$.
- Finally, by the rule (PE-TYPE CASE T), we conclude $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}$.

(II) $\dfrac{e_2 \rightsquigarrow e_2'}{e \rightsquigarrow e_1 \in s \,?\, e_2' : e_3}$ where $e' = e_1 \in s \,?\, e_2' : e_3$:
- From (3), there are two cases to consider.
- First, consider the case for the rule (PE-TYPE CASE T). Then, we have the following assumptions:
$$(a)\ \mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow v_{\mathsf{p}0} \quad (b)\ v_{\mathsf{p}0} \in_{\mathsf{p}} s \quad (c)\ \mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_2' \Downarrow v_{\mathsf{p}}$$
By IH on $e_2 \rightsquigarrow e_2'$ with (c), we have $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_2 \Downarrow {v_{\mathsf{p}}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} {v_{\mathsf{p}}}'$. Then, by the rule (PE-TYPE CASE T), we complete the proof.
- Now consider the case for the rule (PE-TYPE CASE F). Then, we have the following assumptions:
$$(a)\ \mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow v_{\mathsf{p}0} \quad (b)\ v_{\mathsf{p}0} \notin_{\mathsf{p}} s \quad (c)\ \mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_3 \Downarrow v_{\mathsf{p}}$$
Then, we just complete the proof by the rule (PE-TYPE CASE F).

(III) $\dfrac{e_3 \rightsquigarrow e_3'}{e \rightsquigarrow e_1 \in s \,?\, e_2 : e_3'}$ where $e' = e_1 \in s \,?\, e_2 : e_3'$:
- Similar to the case for (II).

(IV) $e_1 = v,\ \vdash v : s$, and $e' = e_2$:
- We have $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow \mathsf{clos}(v)$.
- By Lemma D.10 with $\vdash v : s$, we have $\mathsf{clos}(v) \in_{\mathsf{p}} s$.
- Then, by the rule (PE-TYPE CASE T) with (3), we conclude $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}$.

(V) $e_1 = v,\ \not\vdash v : s$, and $e' = e_3$:
- We have $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow \mathsf{clos}(v)$.
- By Lemma D.11 with $\not\vdash v : s$, we have $\mathsf{clos}(v) \notin_{\mathsf{p}} s$.
- Then, by the rule (PE-TYPE CASE F) with (3), we conclude $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}$.

($e = e_0\sigma_J$): $e' = e_0@\sigma_J$.
- By Lemma D.16 with (3), we have $\sigma_J; \varnothing \vdash_{\mathsf{p}} e_0 \Downarrow {v_{\mathsf{p}}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} {v_{\mathsf{p}}}'$.
- By the rule (PE-INSTANCE), we conclude $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_0\sigma_J \Downarrow {v_{\mathsf{p}}}'$.

($e = \texttt{let}\ \underline{x}\ \texttt{=}\ e_1\ \texttt{in}\ e_2$):

(I) $\dfrac{e_1 \rightsquigarrow e_1'}{e \rightsquigarrow \texttt{let}\ \underline{x}\ \texttt{=}\ e_1'\ \texttt{in}\ e_2}$ where $e' = \texttt{let}\ \underline{x}\ \texttt{=}\ e_1'\ \texttt{in}\ e_2$:
- From (3), we have the following assumptions:
$$(a)\ \mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1' \Downarrow v_{\mathsf{p}0} \quad (b)\ \mathbb{i}; \underline{x} \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} e_2 \Downarrow v_{\mathsf{p}}$$
- By IH on $e_1 \rightsquigarrow e_1'$ with (a), we have $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow {v_{\mathsf{p}0}}'$ and $v_{\mathsf{p}0} \equiv_{\mathsf{c}} {v_{\mathsf{p}0}}'$.
- By Lemma D.15 with (b), we have $\mathbb{i}; \underline{x} \mapsto {v_{\mathsf{p}0}}' \vdash_{\mathsf{p}} e_2 \Downarrow {v_{\mathsf{p}}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} {v_{\mathsf{p}}}'$.
- Then, by the rule (PE-LET), we conclude $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow {v_{\mathsf{p}}}'$.

(II) $\dfrac{e_2 \rightsquigarrow e_2'}{e \rightsquigarrow \texttt{let}\ \underline{x}\ \texttt{=}\ e_1\ \texttt{in}\ e_2'}$ where $e' = \texttt{let}\ \underline{x}\ \texttt{=}\ e_1\ \texttt{in}\ e_2'$:
- From (3), we have the following assumptions:
$$(a)\ \mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow v_{\mathsf{p}0} \quad (b)\ \mathbb{i}; \underline{x} \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} e_2' \Downarrow v_{\mathsf{p}}$$
- By IH on $e_2 \rightsquigarrow e_2'$ with (b), we have $\mathbb{i}; \underline{x} \mapsto v_{\mathsf{p}0} \vdash_{\mathsf{p}} e_2 \Downarrow {v_{\mathsf{p}}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} {v_{\mathsf{p}}}'$.
- Then, by the rule (PE-LET), we conclude $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow {v_{\mathsf{p}}}'$.

(III) $e_1 = v$ and $e' = e_2\{{}^{v}/_{\underline{x}}\}$:
- We have $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e_1 \Downarrow \mathsf{clos}(v)$.
- By Lemma D.17 with (3), we have $\mathbb{i}; \underline{x} \mapsto \mathsf{clos}(v) \vdash_{\mathsf{p}} e_2 \Downarrow {v_{\mathsf{p}}}'$ and $v_{\mathsf{p}} \equiv_{\mathsf{c}} {v_{\mathsf{p}}}'$.
- Finally, by the rule (PE-LET), we conclude $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow {v_{\mathsf{p}}}'$.

□

Lemma D.19. *If* $\vdash e : t$ *and* $e \rightsquigarrow^* v$, *then* $\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow v_{\mathsf{p}}$ *and* $v = (\!|v_{\mathsf{p}}|\!)$.

Proof. By induction on $e \rightsquigarrow^* v$. Suppose $e = v$. Then $v_\mathsf{p} = \mathsf{clos}(v)$ and $(\!|v_\mathsf{p}|\!) = v$.

Now suppose $e \rightsquigarrow e_1 \rightsquigarrow^* v$. By induction hypothesis on $e_1 \rightsquigarrow^* v$, we have $\mathbb{i}; \varnothing \vdash_\mathsf{p} e_1 \Downarrow v_\mathsf{p}'$ and $v = (\!|v_\mathsf{p}'|\!)$. Then, by Lemma D.18, we have $\mathbb{i}; \varnothing \vdash_\mathsf{p} e \Downarrow v_\mathsf{p}$ and $v_\mathsf{p} \equiv_\mathsf{c} v_\mathsf{p}'$, and therefore $v = (\!|v_\mathsf{p}|\!)$. □

Theorem D.20. *Let e be a well-typed closed explicitly-typed expression ($\vdash e : t$). Then*

$$\mathbb{i}; \varnothing \vdash_\mathsf{p} e \Downarrow v \iff e \rightsquigarrow^* (\!|v|\!)$$

Proof. Follows from Lemmas D.12 and D.19. □

D.2 Compilation of the polymorphic language into the intermediate language

In this section, we prove the adequacy of the compilation of the polymorphic language into the intermediate language defined in Section 5.3 (extended with let-polymorphism in Section 5.4). For clarity, in this section, we write respectively e_o, v_o, and $\mathscr{E}_\mathsf{o}$ for the expressions, values, and environments for the intermediate language.

Definition D.21 (Intermediate language)**.**

$$\begin{array}{rcl} e_\mathsf{o} & ::= & c \mid x \mid x_\Sigma \mid \lambda^t_\Sigma x.e_\mathsf{o} \mid e_\mathsf{o} e_\mathsf{o} \mid e_\mathsf{o} \in t \mathbin{?} e_\mathsf{o} : e_\mathsf{o} \mid \mathtt{let}\ \underline{x} = e_\mathsf{o}\ \mathtt{in}\ e_\mathsf{o} \\ v_\mathsf{o} & ::= & c \mid \langle \lambda^t_\Sigma x.e_\mathsf{o}, \mathscr{E}_\mathsf{o} \rangle \\ \Sigma & ::= & \sigma_I \mid \mathtt{comp}(\Sigma, \Sigma') \mid \mathtt{sel}(x, t, \Sigma) \mid \langle \mathscr{E}_\mathsf{o}, \Sigma \rangle \end{array}$$

In this definition, we added a new symbolic substitution of the form $\langle \mathscr{E}_\mathsf{o}, \Sigma \rangle$, which is absent from Section 5.3. Furthermore, we use the following new evaluation rule for polymorphic let-variables:

$$\text{(OE-PVAR}_f\text{)}\quad \frac{\mathscr{E}_\mathsf{o}(\underline{x}) = \langle \lambda^t_{\Sigma'} y.e_\mathsf{o}, \mathscr{E}_\mathsf{o}' \rangle}{\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} x_\Sigma \Downarrow \langle \lambda^t_{\mathtt{comp}(\langle \mathscr{E}_\mathsf{o}, \Sigma \rangle, \Sigma')} y.e_\mathsf{o}, \mathscr{E}_\mathsf{o}' \rangle}$$

The difference from the previous rule is that in this rule we use in the decoration $\langle \mathscr{E}_\mathsf{o}, \Sigma \rangle$ instead of Σ. The main reason is that Σ in x_Σ may contain a symbolic substitution of the form $\mathtt{sel}(y, t, \Sigma_0)$ such that $y \notin \mathsf{dom}(\mathscr{E}_\mathsf{o}')$. Such a symbolic substitution as $\mathtt{sel}(y, t, \Sigma_0)$ may be generated if the let-variable $\underline{x}$ is used inside λ-abstraction in the body of the let-expression. However, in practice, only type-substitutions for the type variables introduced before or in the let-binding may be applied to the polymorphic let-variable, which are already recorded in the closure for the let-variable. Therefore, for the implementation, it is safe to ignore such substitutions as $\langle \mathscr{E}_\mathsf{o}, \Sigma \rangle$ without evaluating them. Still, introducing this new extra symbolic substitution makes the intermediate language and its evaluation semantics clearer.

Definition D.22 (Compilation)**.**

$$\begin{array}{rcl} [\![c]\!]_\Sigma & = & c \\ [\![x]\!]_\Sigma & = & x \\ [\![\underline{x}]\!]_\Sigma & = & x_\Sigma \\ [\![\lambda^t_{\sigma_I} x.e]\!]_\Sigma & = & \lambda^t_{\mathtt{comp}(\Sigma, \sigma_I)} x.[\![e]\!]_{\mathtt{sel}(x, t, \mathtt{comp}(\Sigma, \sigma_I))} \\ [\![e_1 e_2]\!]_\Sigma & = & [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma \\ [\![e\sigma_I]\!]_\Sigma & = & [\![e]\!]_{\mathtt{comp}(\Sigma, \sigma_I)} \\ [\![e_1 \in t \mathbin{?} e_2 : e_3]\!]_\Sigma & = & [\![e_1]\!]_\Sigma \in t \mathbin{?} [\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma \\ [\![\mathtt{let}\ \underline{x} = e_1\ \mathtt{in}\ e_2]\!]_\Sigma & = & \mathtt{let}\ \underline{x} = [\![e_1]\!]_\Sigma\ \mathtt{in}\ [\![e_2]\!]_\Sigma \end{array}$$

Definition D.23 (Membership)**.**

$$\begin{array}{rcl} c \in_\mathsf{o} t & \overset{\text{def}}{\iff} & b_c \leq t \\ \langle \lambda^s_\Sigma x.e_\mathsf{o}, \mathscr{E}_\mathsf{o} \rangle \in_\mathsf{o} t & \overset{\text{def}}{\iff} & s(\mathit{eval}(\mathscr{E}_\mathsf{o}, \Sigma)) \leq t \end{array}$$

where the evaluation of the symbolic set of type-substitutions is inductively defined as

$$\begin{array}{rcl} \mathit{eval}(\mathscr{E}_\mathsf{o}, \sigma_I) & = & \sigma_I \\ \mathit{eval}(\mathscr{E}_\mathsf{o}, \mathtt{comp}(\Sigma, \Sigma')) & = & \mathit{eval}(\mathscr{E}_\mathsf{o}, \Sigma) \circ \mathit{eval}(\mathscr{E}_\mathsf{o}, \Sigma') \\ \mathit{eval}(\mathscr{E}_\mathsf{o}, \mathtt{sel}(x, \bigwedge_{i \in I} t_i \to s_i, \Sigma)) & = & [\ \sigma_j \in \mathit{eval}(\mathscr{E}_\mathsf{o}, \Sigma) \mid \exists i \in I : \mathscr{E}_\mathsf{o}(x) \in_\mathsf{o} t_i \sigma_j\ ] \\ \mathit{eval}(\mathscr{E}_\mathsf{o}, \langle \mathscr{E}_\mathsf{o}', \Sigma \rangle) & = & \mathit{eval}(\mathscr{E}_\mathsf{o}, \Sigma) \end{array}$$

Definition D.24. *Let $\mathscr{E}$ and v_p be an environment and a value for the polymorphic language and $\mathscr{E}_\mathsf{o}$ and v_o for the intermediate language. We define an equivalence relation $\equiv_\mathsf{o}$ between them as follows:*

$$\begin{array}{rcl} \mathscr{E} \equiv_\mathsf{o} \mathscr{E}_\mathsf{o} & \overset{\text{def}}{\iff} & \mathsf{dom}(\mathscr{E}) = \mathsf{dom}(\mathscr{E}_\mathsf{o}) \text{ and } \forall x \in \mathsf{dom}(\mathscr{E}), \mathscr{E}(x) \equiv_\mathsf{o} \mathscr{E}_\mathsf{o}(x) \\ & & \text{and } \forall \underline{x} \in \mathsf{dom}(\mathscr{E}), \mathscr{E}(\underline{x}) \equiv_\mathsf{o} \mathscr{E}_\mathsf{o}(\underline{x}) \\ \langle \lambda^t_{\sigma_K} x.e, \mathscr{E}, \sigma_H \rangle \equiv_\mathsf{o} \langle \lambda^t_\Sigma x.e_\mathsf{o}, \mathscr{E}_\mathsf{o} \rangle & \overset{\text{def}}{\iff} & \sigma_H \circ \sigma_K = \mathit{eval}(\mathscr{E}_\mathsf{o}, \Sigma) \text{ and } e_\mathsf{o} = [\![e]\!]_{\mathtt{sel}(x, t, \Sigma)} \text{ and } \mathscr{E} \equiv_\mathsf{o} \mathscr{E}_\mathsf{o} \end{array}$$

Moreover, for any constant c, by definition $c \equiv_\mathsf{o} c$.

Note that if $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$, then $(\!|v_\mathsf{p}|\!) = (\!|v_\mathsf{o}|\!)$, but not vice versa.

Lemma D.25. *Let $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$. Then:*

(1) $v_\mathsf{p} \in_\mathsf{p} t$ if and only if $v_\mathsf{o} \in_\mathsf{o} t$.

(2) $v_\mathsf{p} \notin_\mathsf{p} t$ if and only if $v_\mathsf{o} \notin_\mathsf{o} t$.

Proof. If $v_\mathsf{p} = v_\mathsf{o} = c$, the proof is trivial. Suppose $v_\mathsf{p} = \langle \lambda^s_{\sigma_K} x.e, \mathscr{E}, \sigma_H \rangle$ and $v_\mathsf{o} = \langle \lambda^s_\Sigma x.e_\mathsf{o}, \mathscr{E}_\mathsf{o} \rangle$. By definition, $v_\mathsf{p} \in_\mathsf{p} t$ if and only if $s(\sigma_H \circ \sigma_K) \leq t$ and $v_\mathsf{o} \in_\mathsf{o} t$ if and only if $s(\mathsf{eval}(\mathscr{E}_\mathsf{o}, \Sigma)) \leq t$. From $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$, we have $\sigma_H \circ \sigma_K = \mathsf{eval}(\mathscr{E}_\mathsf{o}, \Sigma)$, thus completing the proof. □

Lemma D.26. *Suppose $\mathscr{E} \equiv_\mathsf{o} \mathscr{E}_\mathsf{o}$ and $\sigma_I = \mathsf{eval}(\mathscr{E}_\mathsf{o}, \Sigma)$ and $\vdash (e@\sigma_I)\langle\!\langle \mathscr{E} \rangle\!\rangle : t$. Then, $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e \Downarrow v_\mathsf{p}$ if and only if $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e]\!]_\Sigma \Downarrow v_\mathsf{o}$ where $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$.*

Proof. (Only-if part) By induction on a derivation of $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e \Downarrow v_\mathsf{p}$.

(PE-CONST): $e = [\![e]\!]_\Sigma = v_\mathsf{p} = v_\mathsf{o} = c$.
(PE-VAR): $e = [\![e]\!]_\Sigma = x$ and $v_\mathsf{p} = \mathscr{E}(x)$ and $v_\mathsf{o} = \mathscr{E}_\mathsf{o}(x)$. From $\mathscr{E} \equiv_\mathsf{o} \mathscr{E}_\mathsf{o}$, we have $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$.
(PE-CLOSURE): $e = \lambda^t_{\sigma_K} x.e'$ and $[\![e]\!]_\Sigma = \lambda^t_{\mathtt{comp}(\Sigma,\sigma_K)} x.[\![e']\!]_{\mathtt{sel}(x,t,\mathtt{comp}(\Sigma,\sigma_K))}$.

- $v_\mathsf{p} = \langle \lambda^t_{\sigma_K} x.e', \mathscr{E}, \sigma_I \rangle$.
- $v_\mathsf{o} = \langle \lambda^t_{\mathtt{comp}(\Sigma,\sigma_K)} x.[\![e']\!]_{\mathtt{sel}(x,t,\mathtt{comp}(\Sigma,\sigma_K))}, \mathscr{E}_\mathsf{o} \rangle$.
- $\mathsf{eval}(\mathscr{E}_\mathsf{o}, \mathtt{comp}(\Sigma, \sigma_K)) = \mathsf{eval}(\mathscr{E}_\mathsf{o}, \Sigma) \circ \mathsf{eval}(\mathscr{E}_\mathsf{o}, \sigma_K) = \sigma_I \circ \sigma_K$ and therefore $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$ by Definition D.24.

(PE-APPLY): $e = e_1 e_2$ and $[\![e]\!]_\Sigma = [\![e_1]\!]_\Sigma [\![e_2]\!]_\Sigma$.

- From $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e \Downarrow v_\mathsf{p}$, we have the following assumptions:
  (a) $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e_1 \Downarrow \langle \lambda^s_{\sigma_K} x.e', \mathscr{E}', \sigma_H \rangle$ where $s = \wedge_{i \in I} s_i \to s'_i$;
  (b) $\sigma_I; \mathscr{E}_1 \vdash_\mathsf{p} e_2 \Downarrow {v_\mathsf{p}}'$;
  (c) $\sigma_J = \sigma_H \circ \sigma_K$ and $P = \{j \in J \mid \exists i \in I : {v_\mathsf{p}}' \in_\mathsf{p} s_i \sigma_j\}$; and
  (d) $\sigma_P; \mathscr{E}', x \mapsto {v_\mathsf{p}}' \vdash_\mathsf{p} e' \Downarrow v_\mathsf{p}$.
- By IH on (a), $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e_1]\!]_\Sigma \Downarrow \langle \lambda^s_{\Sigma'} x.e_\mathsf{o}, {\mathscr{E}_\mathsf{o}}' \rangle$ where $\mathsf{eval}({\mathscr{E}_\mathsf{o}}', \Sigma') = \sigma_H \circ \sigma_K = \sigma_J$ and $\mathscr{E}' \equiv_\mathsf{o} {\mathscr{E}_\mathsf{o}}'$ and $e_\mathsf{o} = [\![e']\!]_{\mathtt{sel}(x,s,\Sigma')}$.
- By IH on (b), $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e_2]\!]_\Sigma \Downarrow {v_\mathsf{o}}'$ where ${v_\mathsf{p}}' \equiv_\mathsf{o} {v_\mathsf{o}}'$.
- From $\mathscr{E}' \equiv_\mathsf{o} {\mathscr{E}_\mathsf{o}}'$ and ${v_\mathsf{p}}' \equiv_\mathsf{o} {v_\mathsf{o}}'$, we have $(\mathscr{E}', x \mapsto {v_\mathsf{p}}') \equiv_\mathsf{o} ({\mathscr{E}_\mathsf{o}}', x \mapsto {v_\mathsf{o}}')$.
- $\mathsf{eval}(({\mathscr{E}_\mathsf{o}}', x \mapsto {v_\mathsf{o}}'), \mathtt{sel}(x, s, \Sigma')) = [\ \sigma_j \in \mathsf{eval}({\mathscr{E}_\mathsf{o}}', \Sigma') \mid \exists i \in I : {v_\mathsf{o}}' \in_\mathsf{o} s_i \sigma_j\ ] = [\ \sigma_j \in \sigma_J \mid \exists i \in I : {v_\mathsf{o}}' \in_\mathsf{o} s_i \sigma_j\ ]$, which is equal to σ_P by Lemma D.25.
- By IH on (d), ${\mathscr{E}_\mathsf{o}}', x \mapsto {v_\mathsf{o}}' \vdash_\mathsf{o} e_\mathsf{o} \Downarrow v_\mathsf{o}$ and $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$.
- Finally, by the rule (OE-APPLY), we conclude $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e]\!]_\Sigma \Downarrow v_\mathsf{o}$.

(PE-TYPE CASE T): $e = e_1 \in t\ ?\ e_2 : e_3$ and $[\![e]\!]_\Sigma = [\![e_1]\!]_\Sigma \in t\ ?\ [\![e_2]\!]_\Sigma : [\![e_3]\!]_\Sigma$.

- From $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e \Downarrow v_\mathsf{p}$, we have the following assumptions:
  (a) $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e_1 \Downarrow {v_\mathsf{p}}'$ (b) ${v_\mathsf{p}}' \in_\mathsf{p} t$ (c) $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e_2 \Downarrow v_\mathsf{p}$
- By IH on (a), $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e_1]\!]_\Sigma \Downarrow {v_\mathsf{o}}'$ and ${v_\mathsf{p}}' \equiv_\mathsf{o} {v_\mathsf{o}}'$.
- By Lemma D.25 with (b), we have ${v_\mathsf{o}}' \in_\mathsf{o} t$.
- By IH on (c), $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e_2]\!]_\Sigma \Downarrow v_\mathsf{o}$ and $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$.
- Finally, by the rule (OE-TYPE CASE T), we conclude $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e]\!]_\Sigma \Downarrow v_\mathsf{o}$.

(PE-TYPE CASE F): Similar to the case for (PE-TYPE CASE T).
(PE-INSTANCE): $e = e_0 \sigma_J$ and $[\![e]\!]_\Sigma = [\![e_0]\!]_{\mathtt{comp}(\Sigma,\sigma_J)}$.

- From $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e \Downarrow v_\mathsf{p}$, we have the following assumption:
  (a) $\sigma_I \circ \sigma_J; \mathscr{E} \vdash_\mathsf{p} e_0 \Downarrow v_\mathsf{p}$
- $\mathsf{eval}(\mathscr{E}_\mathsf{o}, \mathtt{comp}(\Sigma, \sigma_J)) = \mathsf{eval}(\mathscr{E}_\mathsf{o}, \Sigma) \circ \mathsf{eval}(\mathscr{E}_\mathsf{o}, \sigma_J) = \sigma_I \circ \sigma_J$.
- By IH on (a), $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e_0]\!]_{\mathtt{comp}(\Sigma,\sigma_J)} \Downarrow v_\mathsf{o}$ and $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$.
- Then, from $[\![e]\!]_\Sigma = [\![e_0]\!]_{\mathtt{comp}(\Sigma,\sigma_J)}$, we complete the proof.

(PE-VAR$_c$): Similar to the case for (PE-CONST).
(PE-VAR$_f$): $e = \underline{x}$ and $[\![e]\!]_\Sigma = x_\Sigma$ and $\mathscr{E}(\underline{x}) = \langle \lambda^t_{\sigma_K} y.e_0, \mathscr{E}', \sigma_H \rangle$.

- From $\mathscr{E} \equiv_\mathsf{o} \mathscr{E}_\mathsf{o}$, we have $\mathscr{E}_\mathsf{o}(\underline{x}) = \langle \lambda^t_{\Sigma'} y.e_\mathsf{o}, {\mathscr{E}_\mathsf{o}}' \rangle$ where $\mathsf{eval}({\mathscr{E}_\mathsf{o}}', \Sigma') = \sigma_H \circ \sigma_K$ and $\mathscr{E}' = {\mathscr{E}_\mathsf{o}}'$ and $e_\mathsf{o} = [\![e_0]\!]_{\mathtt{sel}(y,t,\Sigma')}$.
- Then, $v_\mathsf{p} = \langle \lambda^t_{\sigma_K} y.e_0, \mathscr{E}', \sigma_I \circ \sigma_H \rangle$ and $v_\mathsf{o} = \langle \lambda^t_{\mathtt{comp}(\langle \mathscr{E}_\mathsf{o}, \Sigma \rangle, \Sigma')} y.e_\mathsf{o}, {\mathscr{E}_\mathsf{o}}' \rangle$.
- $\mathsf{eval}({\mathscr{E}_\mathsf{o}}', \mathtt{comp}(\langle \mathscr{E}_\mathsf{o}, \Sigma \rangle, \Sigma')) = \mathsf{eval}({\mathscr{E}_\mathsf{o}}', \langle \mathscr{E}_\mathsf{o}, \Sigma \rangle) \circ \mathsf{eval}({\mathscr{E}_\mathsf{o}}', \Sigma') = \mathsf{eval}(\mathscr{E}_\mathsf{o}, \Sigma) \circ (\sigma_H \circ \sigma_K) = \sigma_I \circ (\sigma_H \circ \sigma_K)$ and therefore $v_\mathsf{p} \equiv v_\mathsf{o}$.

(PE-LET): $e = \mathtt{let}\ \underline{x} = e_1\ \mathtt{in}\ e_2$ and $[\![e]\!]_\Sigma = \mathtt{let}\ \underline{x} = [\![e_1]\!]_\Sigma\ \mathtt{in}\ [\![e_2]\!]_\Sigma$.

- From $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e \Downarrow v_\mathsf{p}$, we have the following assumptions:
  (a) $\sigma_I; \mathscr{E} \vdash_\mathsf{p} e_1 \Downarrow {v_\mathsf{p}}'$ (b) $\sigma_I; \mathscr{E}, \underline{x} \mapsto {v_\mathsf{p}}' \vdash_\mathsf{p} e_2 \Downarrow v_\mathsf{p}$
- By IH on (a), $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e_1]\!]_\Sigma \Downarrow {v_\mathsf{o}}'$ and ${v_\mathsf{p}}' \equiv_\mathsf{o} {v_\mathsf{o}}'$.
- By IH on (b), $\mathscr{E}_\mathsf{o}, \underline{x} \mapsto {v_\mathsf{o}}' \vdash_\mathsf{o} [\![e_1]\!]_\Sigma \Downarrow v_\mathsf{o}$ and $v_\mathsf{p} \equiv_\mathsf{o} v_\mathsf{o}$.
- Finally, by the rule (OE-LET), we conclude $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e]\!]_\Sigma \Downarrow v_\mathsf{o}$.

(If part) By induction on a derivation of $\mathscr{E}_\mathsf{o} \vdash_\mathsf{o} [\![e]\!]_\Sigma \Downarrow v_\mathsf{o}$. The proof is similar to the proof for the (only-if part). □

Theorem D.27. *Let e be a well-typed closed explicitly-typed expression ($\vdash_{\mathscr{A}} e : t$). Then*

$$\mathbb{i}; \varnothing \vdash_{\mathsf{p}} e \Downarrow v \iff \vdash_{\mathsf{o}} [\![e]\!]_{\mathbb{i}} \Downarrow v'$$

with $(\!|v|\!) = (\!|v'|\!)$.

Proof. Follows from Lemma D.26. □

Corollary D.28 (Adequacy of the compilation)**.** *Let e be a well-typed closed explicitly-typed expression ($\vdash_{\mathscr{A}} e : t$). Then*

$$\vdash_{\mathsf{o}} [\![e]\!]_{\mathbb{i}} \Downarrow v_{\mathsf{o}} \iff e \rightsquigarrow^* (\!|v|\!)$$

Proof. Follows from Theorems D.20 and D.27. □

E. Syntactic sugar for type-case

In order to make type-case closer to pattern matching, it is handy to define an extension of the type-case expression that features binding, which we write $(x{=}e){\in}t\,?\,e_1 : e_2$, and that can be encoded as:

$$(\lambda^{((s\wedge t)\to t_1)\wedge((s\wedge\neg t)\to t_2)} x.x{\in}t\,?\,e_1 : e_2)e$$

where s is the type of e, t_1 the type of e_1, and t_2 the type of e_2. We add another twist to this construct and define a particular (purely) syntactic sugar: $x\,\boldsymbol{\epsilon}\,t\,?\,e_1 : e_2$ (notice the boldface "belongs to" symbol) which stands for $(x{=}x){\in}t\,?\,e_1 : e_2$. The reader may wonder what is the interest of binding a variable to itself. Actually, the two occurrences of x in $(x{=}x){\in}t$ denote two distinct variables: the one on the right is recorded in the environment with some type s; this variable does not occur either in e_1 or e_2 because it is hidden by the x on the left; this binds the occurrences of x in e_1 and e_2 but with different types, $s{\wedge}t$ in e_1 and $s{\wedge}\neg t$ in e_2. This allows the system to use different type assumptions for x in each branch, as stated by the (algorithmic) typing rule directly derived from the encoding:

$$\frac{t_i \not\simeq \mathbb{0} \quad \Rightarrow \quad \Delta \fatsemi \Gamma, (x : t_i) \vdash e_i : s_i}{\Delta \fatsemi \Gamma \vdash x\,\boldsymbol{\epsilon}\,t\,?\,e_1 : e_2 \;:\; \bigvee_{t_i \not\simeq \mathbb{0}} s_i} \begin{array}{l} t_1 = \Gamma(x) \wedge t \\ t_2 = \Gamma(x) \wedge \neg t \end{array} \quad (\textit{case-var})$$

Note that x is defined in Γ but its type $\Gamma(x)$ is overridden in the premises to type the branches. We already silently used this construction in the body of the definition of `map` in Section 2 which should have been written with the boldface "belongs to" symbol:

$$\lambda^{[\alpha]\to[\beta]}\ell\,.\,\ell\,\boldsymbol{\epsilon}\,\texttt{nil}\,?\,\texttt{nil} : (f(\pi_1\ell),mf(\pi_2\ell)),$$

since the presence of the π_i's in the second branch needs the hypothesis $\ell : [\alpha]\backslash\texttt{nil}$ (the expression $\pi_i\ell$ is well-typed only if ℓ is a product, that is, in this case it a non-empty list). If we do not use the boldface belong symbol, then each branch (in particular, the second one) is typed under the hypothesis $\ell : [\alpha]$ and, thus, type-check fails. In practice, any real programming language will implement just $\boldsymbol{\epsilon}$ and not $\in$ whenever the tested expression is a variable.